RxJava Internals:part-2

Agenda

- Introduction
- Observables and Observer
- Operators
- Combining Observables
- Multicasting, Replaying and Caching
- Concurrency
- Switching, Throttling, Windowing and buffering
- Flowables and Backpressure

Agenda

- Introduction
- Observables and Observer
- Operators
- **Combining Observables**
- Multicasting, Replaying and Caching
- Concurrency
- Switching, Throttling, Windowing and buffering
- Flowables and Backpressure

Combining

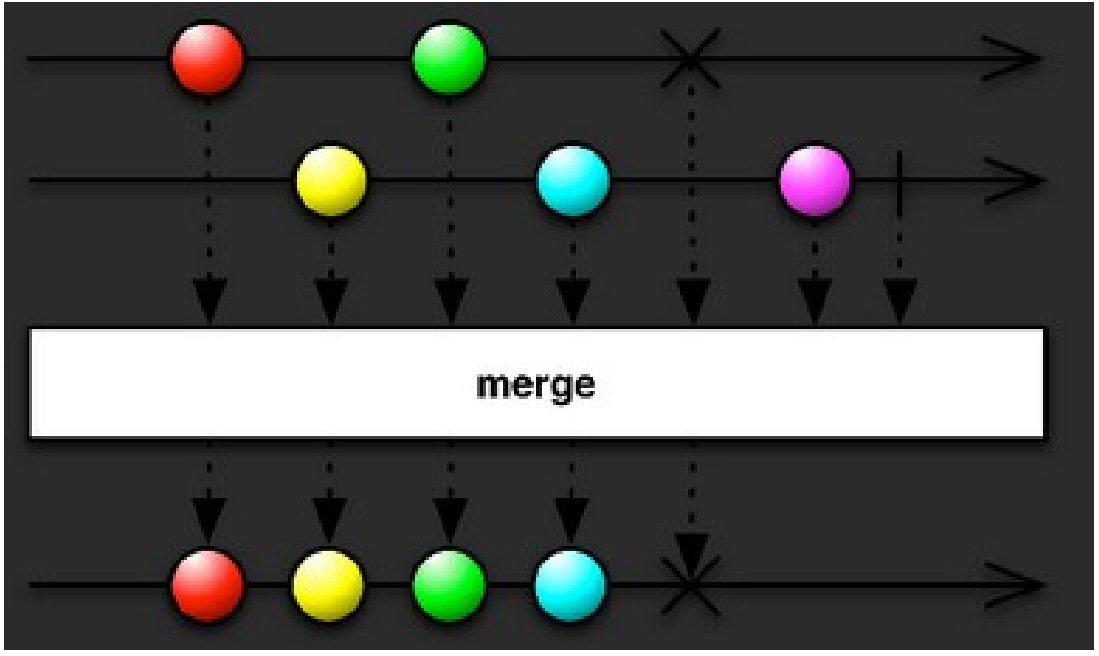
- **Merge:**
  - If we have only two to four Observable<T> sources to merge, you can pass each one as an argument to the Observable.merge() factory

```
public static void main(String[] args) {
    Observable<String> src1 = Observable.just("Alpha", "Beta");
    Observable<String> src2 = Observable.just("Zeta", "Eta");
    Observable.merge(src1, src2)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

```
RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Zeta
RECEIVED: Eta
```

Combining

- **Merge:**



Combining

- **MergeWith:**
  - Alternatively, use mergeWith(), which is the operator version of Observable.merge()

```
public static void main(String[] args) {
    Observable<String> src1 = Observable.just("Alpha", "Beta");
    Observable<String> src2 = Observable.just("Zeta", "Eta");
    src1.mergeWith(src2)
        .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

Combining

- **MergeArray:**
  - If you have more than four Observable<T> sources, you can use Observable.mergeArray() to pass an array of Observable instances that you want to merge.
    - Since RxJava 2.0 was written for JDK 6+ and has no access to a @SafeVarargs annotation

```
public static void main(String[] args) {
    Observable<String> src1 = Observable.just("Alpha", "Beta");
    Observable<String> src2 = Observable.just("Gamma", "Delta");
    Observable<String> src3 = Observable.just("Epsilon", "Zeta");
    Observable<String> src4 = Observable.just("Eta", "Theta");
    Observable<String> src5 = Observable.just("Iota", "Kappa");
    Observable.mergeArray(src1, src2, src3, src4, src5)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

```
RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Gamma
RECEIVED: Delta
RECEIVED: Epsilo
RECEIVED: Zeta
RECEIVED: Eta
RECEIVED: Theta
RECEIVED: Iota
RECEIVED: Kappa
```

Combining

- **MergeList:**
  - There is also an overloaded version of Observable.merge() that accepts Iterable<Observable<T>> and produces the same results in a more type-safe manner.

```
public static void main(String[] args) {
    Observable<String> src1 = Observable.just("Alpha", "Beta");
    Observable<String> src2 = Observable.just("Gamma", "Delta");
    Observable<String> src3 = Observable.just("Epsilon", "Zeta");
    Observable<String> src4 = Observable.just("Eta", "Theta");
    Observable<String> src5 = Observable.just("Iota", "Kappa");
    List<Observable<String>> sources =
            Arrays.asList(src1, src2, src3, src4, src5);
    Observable.merge(sources)
           .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

Combining

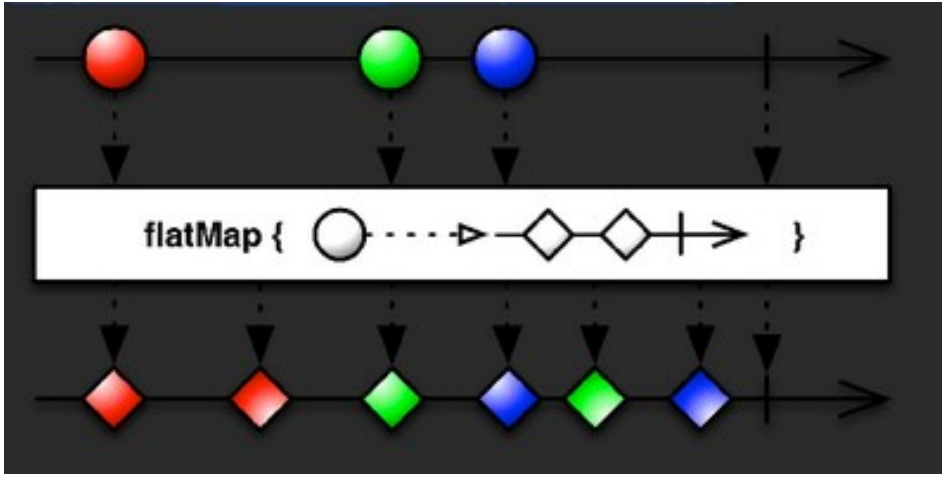
- **Merge Infinite Stream:**
  - Observable.merge() works with infinite observables. Since it will subscribe to all observables and fire their emissions as soon as they are available, you can merge multiple infinite sources into a single stream.

```
public static void main(String[] args) {
    //emit every second
    Observable<String> src1 = Observable.interval( period: 1,
            TimeUnit.SECONDS)
            .map(l -> l + 1) // emit elapsed seconds
            .map(l -> "Source1: " + l + " seconds");
    //emit every 300 milliseconds
    Observable<String> src2 =
            Observable.interval( period: 300, TimeUnit.MILLISECONDS)
                    .map(l -> (l + 1) * 300) // emit elapsed milliseconds
                    .map(l -> "Source2: " + l + " milliseconds");
    //merge and subscribe
    Observable.merge(src1, src2)
            .subscribe(System.out::println);
    //keep alive for 3 seconds
    sleep( millis: 3000);
}
```

```
Source2: 300 milliseconds
Source2: 600 milliseconds
Source2: 900 milliseconds
Source1: 1 seconds
Source2: 1200 milliseconds
Source2: 1500 milliseconds
Source2: 1800 milliseconds
Source1: 2 seconds
Source2: 2100 milliseconds
Source2: 2400 milliseconds
Source2: 2700 milliseconds
Source1: 3 seconds
Source2: 3000 milliseconds
```

Combining

- **flatmap**
  - Returns an Observable that emits items based on applying a function that you supply to each item emitted by the current Observable, where that function returns an ObservableSource, and then merging those returned ObservableSources and emitting the results of this merger.



Combining

- **flatmap**
  - The flatMap() operator is one of, if not the, most powerful operators in RxJava.
    - It performs a dynamic Observable.merge() by taking each emission and **mapping it to an Observable**.
    - Then, it **merges the resulting observables** into a single stream.

Combining

- **flatmap**

```
public static void main(String[] args) {
    Observable<String> source =
            Observable.just("Alpha", "Beta", "Gamma");
    source.flatMap(s -> Observable.fromArray(s.split("")))
            .subscribe(System.out::println);
}
```

```
A
l
p
h
a
B
e
t
a
G
a
m
m
a
```

Combining

- **flatmap**

```
public static void main(String[] args) {
    Observable<String> source =
            Observable.just("521934/2342/FOXTROT",
                            "21962/12112/TANGO/78886");
    source.flatMap(s -> Observable.fromArray(s.split("/")))
          //use regex to filter integers
          .filter(s -> s.matches("[0-9]+"))
          .map(Integer::valueOf)
          .subscribe(System.out::println);
}
```

```
521934
2342
21962
12112
78886
```

Combining

- **Flatmap**
  - the source Observable emits the values 2, 3, 10, and 7, each transformed by flatMap() to an interval Observable that emits a value every 2, 3, 10, and 7 seconds, respectively.
    - The four Observable instances produced by flatMap() are then merged into a single stream:

```
public static void main(String[] args) {
    Observable.just(2, 3, 10, 7)
        .flatMap(i -> Observable.interval(i, TimeUnit.SECONDS)
                        .map(i2 -> i + "s interval: " +
                                ((i + 1) * i) + " seconds elapsed"))
        .subscribe(System.out::println);
    sleep(12000);
}
```

```
2s interval: 2 seconds elapsed
3s interval: 3 seconds elapsed
2s interval: 4 seconds elapsed
2s interval: 6 seconds elapsed
3s interval: 6 seconds elapsed
7s interval: 7 seconds elapsed
2s interval: 8 seconds elapsed
3s interval: 9 seconds elapsed
2s interval: 10 seconds elapsed
10s interval: 10 seconds elapsed
2s interval: 12 seconds elapsed
3s interval: 12 seconds elapsed
```

Combining

- **Flatmap**
  - flatMap() is that it can be used in many clever ways. Evaluate each received value within flatMap() and figure out what kind of Observable you want to return.

```
public static void main(String[] args) {
    Observable.just(2, 0, 3, 10, 7) Observable<Integer>
            .flatMap(i -> {
                if (i == 0){
                    return Observable.empty();
                }
                else {
                    return Observable.interval(i, TimeUnit.SECONDS)
                            .map(l -> i + "s interval: " +
                                    ((l + 1) * i) + " seconds elapsed");
                }
            }) Observable<String>
            .subscribe(System.out::println);
    sleep( millis: 12000);
}
```

Combining

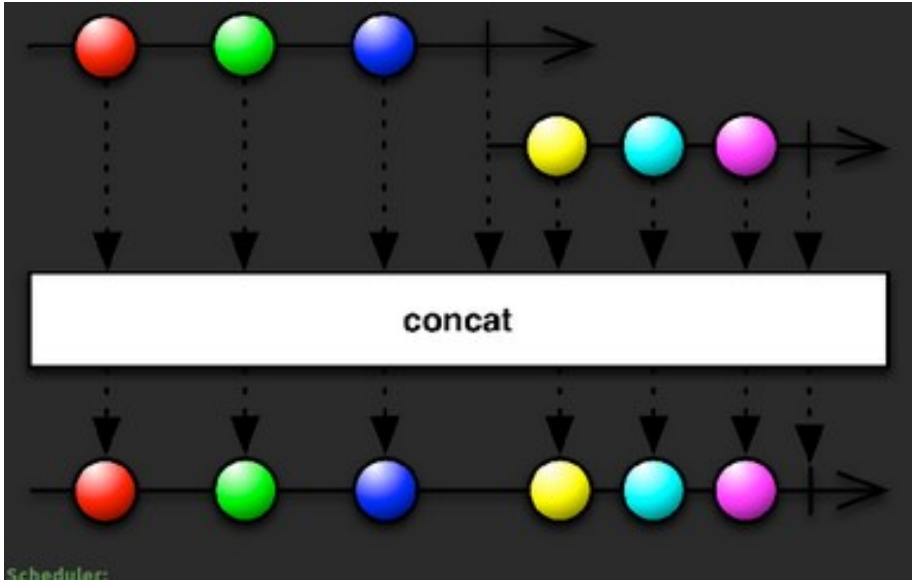
- **Flatmap with combiner:**
  - The flatMap() operator has an overloaded version, flatMap(Function<T,Observable<R>> mapper, BiFunction<T,U,R> combiner), that allows the provision of a combiner along with the mapper function.

```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
            .flatMap(s -> Observable.fromArray(s.split("")),
                                    (s, r) -> s + "-" + r)
            .subscribe(System.out::println);
}
```

```
Alpha-A
Alpha-l
Alpha-p
Alpha-h
Alpha-a
Beta-B
Beta-e
Beta-t
Beta-a
Gamma-G
```

Combining

- **concat**
  - The Observable.concat() factory is the concatenation equivalent to Observable.merge(). It will combine the emitted values of multiple observables, but will fire each one **sequentially** and only move to the next after onComplete() is called.



Combining

- **concat**
  - The Observable.concat() factory is the concatenation equivalent to Observable.merge(). It will combine the emitted values of multiple observables, but will fire each one **sequentially** and only move to the next after onComplete() is called.

```
public static void main(String[] args) {
    Observable<String> src1 = Observable.just("Alpha", "Beta");
    Observable<String> src2 = Observable.just("Zeta", "Eta");
    Observable.concat(src1, src2)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

```
RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Zeta
RECEIVED: Eta
```

Combining

- **concat**
  - The Observable.concat() factory is the concatenation equivalent to Observable.merge(). It will combine the emitted values of multiple observables, but will fire each one **sequentially** and only move to the next after onComplete() is called.

```
public static void main(String[] args) {
    Observable<String> src1 = Observable.just("Alpha", "Beta");
    Observable<String> src2 = Observable.just("Zeta", "Eta");
    Observable.concat(src1, src2)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
}
```

```
RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Zeta
RECEIVED: Eta
```

Combining

- **concat**

```
public static void main(String[] args) {
    //emit every second, but only take 2 emissions
    Observable<String> src1 =
            Observable.interval( period: 1, TimeUnit.SECONDS)
                    .take( count: 2)
                    .map(l -> l + 1) // emit elapsed seconds
                    .map(l -> "Source1: " + l + " seconds");
    //emit every 300 milliseconds
    Observable<String> src2 =
            Observable.interval( period: 300, TimeUnit.MILLISECONDS)
                    .map(l -> (l + 1) * 300) // emit elapsed millis
                    .map(l -> "Source2: " + l + " milliseconds");
    Observable.concat(src1, src2)
            .subscribe(i -> System.out.println("RECEIVED: " + i
    //keep application alive for 5 seconds
    sleep( millis: 5000);
}
```

```
RECEIVED: Source1: 1 seconds
RECEIVED: Source1: 2 seconds
RECEIVED: Source2: 300 milliseconds
RECEIVED: Source2: 600 milliseconds
RECEIVED: Source2: 900 milliseconds
RECEIVED: Source2: 1200 milliseconds
RECEIVED: Source2: 1500 milliseconds
```

Combining

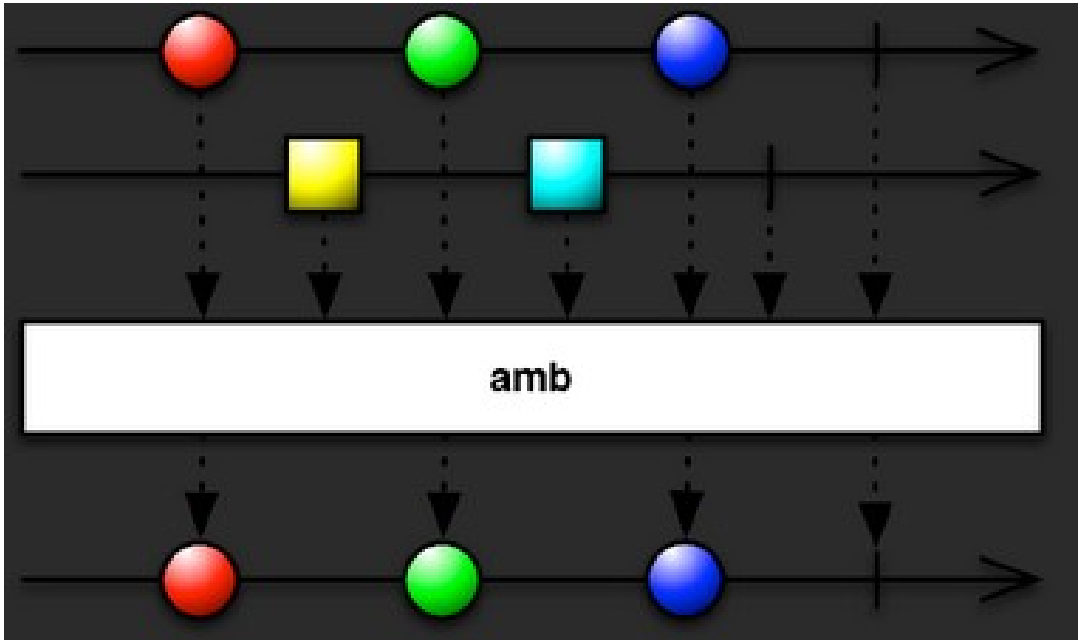
- **ConcatMap:**
  - concatMap() if we explicitly cared about emission order.

```
public static void main(String[] args) {
    Observable<String> source =
                    Observable.just("Alpha", "Beta", "Gamma");
    source.concatMap(s -> Observable.fromArray(s.split("")))
          .subscribe(System.out::println);
}
```

```
A
l
p
h
a
B
e
t
a
G
a
m
m
a
```

Combining

- **Ambigous:**
  - Mirrors the one ObservableSource in an array of several ObservableSources that first either emits an item or sends a termination notification.



Combining

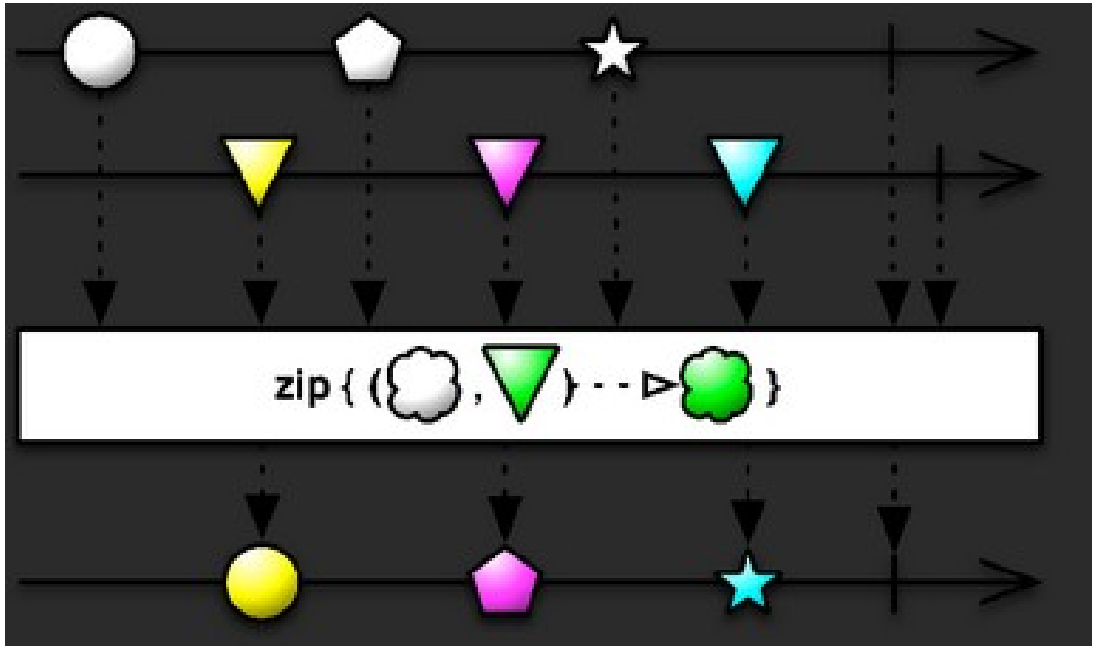
- **Ambigous:**
  - Two interval sources and we combine them with the Observable.amb() factory. If one emits every second while the other emits every 300 milliseconds, the latter is going to win because it emits more often:

```
public static void main(String[] args) {
    //emit every second
    Observable<String> src1 =
          Observable.interval(1, TimeUnit.SECONDS)
                    .take(2)
                    .map(l -> l + 1) // emit elapsed seconds
                    .map(l -> "Source1: " + l + " seconds");
    //emit every 300 milliseconds
    Observable<String> src2 =
          Observable.interval(300, TimeUnit.MILLISECONDS)
                .map(l -> (l + 1) * 300) // emit elapsed millis
                .map(l -> "Source2: " + l + " milliseconds");
    //emit Observable that emits first
    Observable.amb(Arrays.asList(src1, src2))
              subscribe(i -> System.out.println("RECEIVED: " + i));
    //keep application alive for 5 seconds
    sleep(5000);
}
```

```
RECEIVED: Source2: 300 milliseconds
RECEIVED: Source2: 600 milliseconds
RECEIVED: Source2: 900 milliseconds
RECEIVED: Source2: 1200 milliseconds
RECEIVED: Source2: 1500 milliseconds
RECEIVED: Source2: 1800 milliseconds
RECEIVED: Source2: 2100 milliseconds
```

Combining

- **zip:**
  - Returns an Observable that emits the results of a specified combiner function applied to combinations of two items emitted, in sequence, by two other ObservableSources.



Combining

- **zip:**
  - If we have an Observable<String> and an Observable<Integer>, we can zip each String and Integer together in a one-to-one pairing. Then, we can combine them using the BiFunction<String,Integer,String> zipper function.

```
public static void main(String[] args) {
    Observable<String> src1 =
            Observable.just("Alpha", "Beta", "Gamma");
    Observable<Integer> src2 = Observable.range(1, 6);
    Observable.zip(src1, src2, (s, i) -> s + "-" + i)
            .subscribe(System.out::println);
}
```

```
Alpha-1
Beta-2
Gamma-3
```

Combining

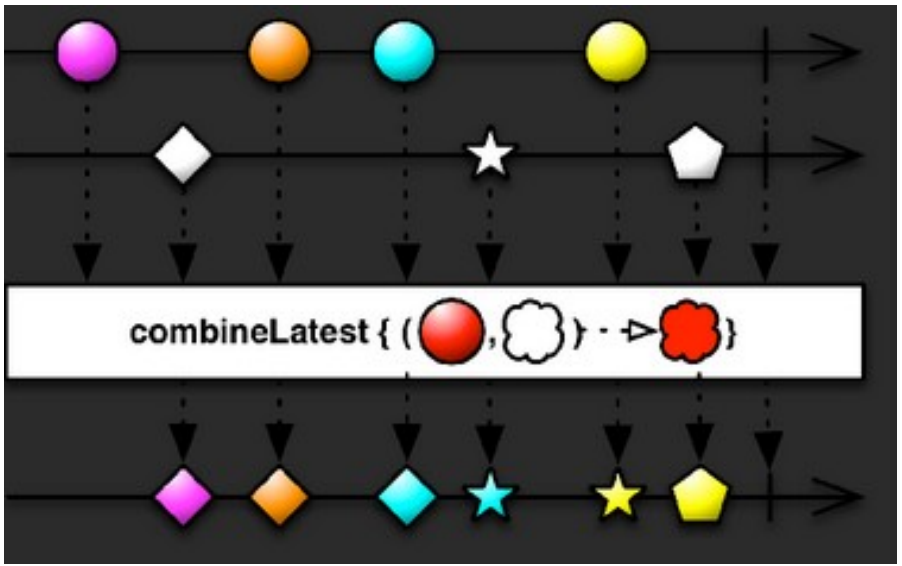
- **zip:**
  - Zipping can also be helpful in slowing down emissions using Observable.interval().
    - Here, we zip each string with a one-second interval.

```
public static void main(String[] args) {
    Observable<String> src1 =
            Observable.just("Alpha", "Beta", "Gamma");
    Observable<Integer> src2 = Observable.range( start: 1,  count: 6);
    Observable.zip(src1, src2, (s, i) -> s + "-" + i)
            .subscribe(System.out::println);
}
```

```
Received Alpha at 13:15:07.038857
Received Beta at 13:15:08.023963
Received Gamma at 13:15:09.018764
```

Combining

- **combineLatest:**
  - If any of the sources never produces an item but only terminates (normally or with an error), the resulting sequence terminates immediately (normally or with all the errors accumulated till that point).
    - If that input source is also synchronous, other sources after it will not be subscribed to.



Combining

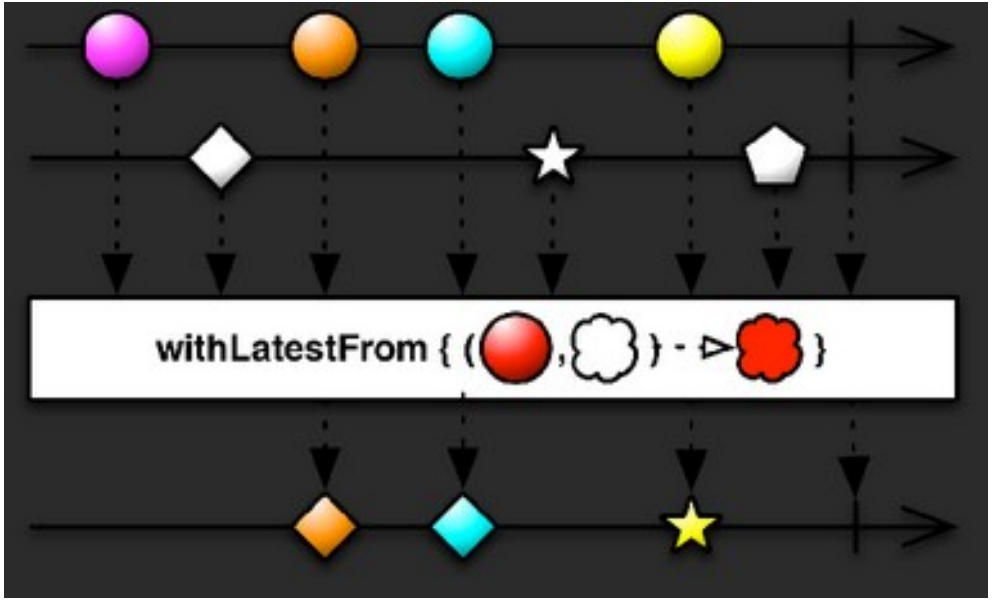
- **combineLatest:**
  - Observable.combineLatest() between two interval observables, the first emitting at 300 milliseconds and the other every second

```
public static void main(String[] args) {
    Observable<Long> source1 =
                    Observable.interval(300, TimeUnit.MILLISECONDS);
    Observable<Long> source2 =
                           Observable.interval(1, TimeUnit.SECONDS);
    Observable.combineLatest(source1, source2, (l1, l2) ->
                            "SOURCE 1: " + l1 + " SOURCE 2: " + l2)
                .subscribe(System.out::println);
    sleep(3000);
}
```

```
SOURCE 1: 2  SOURCE 2: 0
SOURCE 1: 3  SOURCE 2: 0
SOURCE 1: 4  SOURCE 2: 0
SOURCE 1: 5  SOURCE 2: 0
SOURCE 1: 5  SOURCE 2: 1
SOURCE 1: 6  SOURCE 2: 1
SOURCE 1: 7  SOURCE 2: 1
SOURCE 1: 8  SOURCE 2: 1
SOURCE 1: 9  SOURCE 2: 1
SOURCE 1: 9  SOURCE 2: 2
```

Combining

- **withLatestFrom:**
  - Note that this operator doesn't emit anything until all other sources have produced at least one value. \
    - The resulting emission only happens when the current Observable emits (and not when any of the other sources emit, unlike combineLatest).



Combining

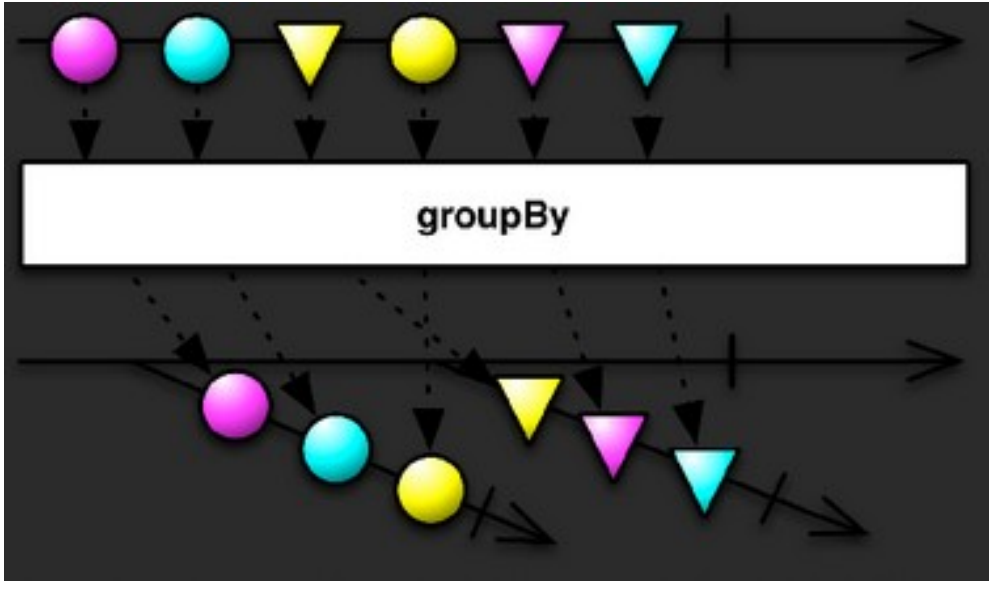
- **withLatestFrom:**

```
public static void main(String[] args) {
    Observable<Long> source1 =
                Observable.interval(300, TimeUnit.MILLISECONDS);
    Observable<Long> source2 =
                        Observable.interval(1, TimeUnit.SECONDS);
    source2.withLatestFrom(source1, (l1, l2) ->
                         "SOURCE 2: " + l1 + " SOURCE 1: " + l2)
           .subscribe(System.out::println);
    sleep(3000);
}
```

```
SOURCE 2: 0  SOURCE 1: 2
SOURCE 2: 1  SOURCE 1: 5
SOURCE 2: 2  SOURCE 1: 9
```

Combining

- **groupBy:**
  - Groups the items emitted by the current Observable according to a specified criterion, and emits these grouped items as GroupedObservables.



Combining

- **groupBy:**
  - we can use the groupBy() operator to group emissions for an Observable<String> by each string's length. We will subscribe to it in a moment, but here is how we declare it:

```
public static void main(String[] args) {
    Observable<String> source = Observable.just("Alpha", "Beta",
                                        "Gamma", "Delta", "Epsilon");
    Observable<GroupedObservable<Integer, String>> byLengths =
                                      source.groupBy(s -> s.length());
    byLengths.flatMapSingle(grp -> grp.toList())
             .subscribe(System.out::println);
}
```

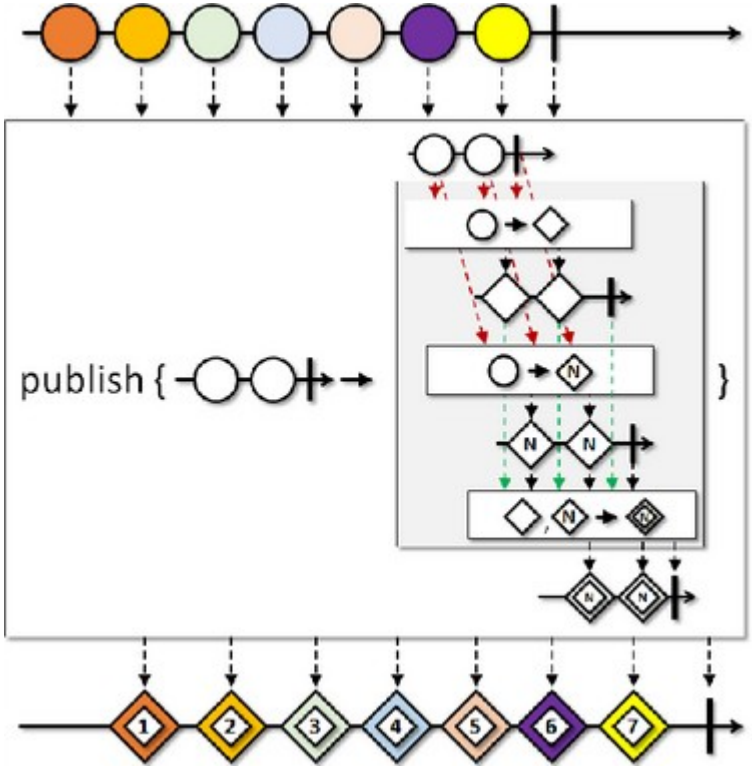
```
[Beta]
[Alpha, Gamma, Delta]
[Epsilon]
```

Agenda

- Introduction
- Observables and Observer
- Operators
- Combining Observables
- **Multicasting, Replaying and Caching(MRC)**
- Concurrency
- Switching, Throttling, Windowing and buffering
- Flowables and Backpressure

MRC

- **multicast:**
  - Returns an Observable that emits the results of invoking a specified selector on items emitted by a ConnectableObservable that shares a single subscription to the current Observable sequence.



MRC

- **multicast:**
  - A cold Observable, such as the one created by Observable.range(), regenerates emissions for each subscribed Observer.

```
public static void main(String[] args) {
    Observable<Integer> ints = Observable.range(1, 3);
    ints.subscribe(i -> System.out.println("Observer One: " + i));
    ints.subscribe(i -> System.out.println("Observer Two: " + i));
}
```

```
Observer One: 1
Observer One: 2
Observer One: 3
Observer Two: 1
Observer Two: 2
Observer Two: 3
```

MRC

- **multicast:**
  - If we wanted to consolidate multiple observables into a single stream of data that pushes each emission to both subscribed observers simultaneously, we can call publish() on Observable, which will return a ConnectableObservable.

```
public static void main(String[] args) {
    ConnectableObservable<Integer> ints =
            Observable.range( start: 1,  count: 3).publish();
    ints.subscribe(i -> System.out.println("Observer One:" + i));
    ints.subscribe(i -> System.out.println("Observer Two:" + i));
    ints.connect();
}
```

```
Observer One: 1
Observer Two: 1
Observer One: 2
Observer Two: 2
Observer One: 3
Observer Two: 3
```

MRC

- **multicastWithOperators:**
  - How multicasting works within a chain of operators, we are going to use Observable.range() and then map each emission to a random integer.

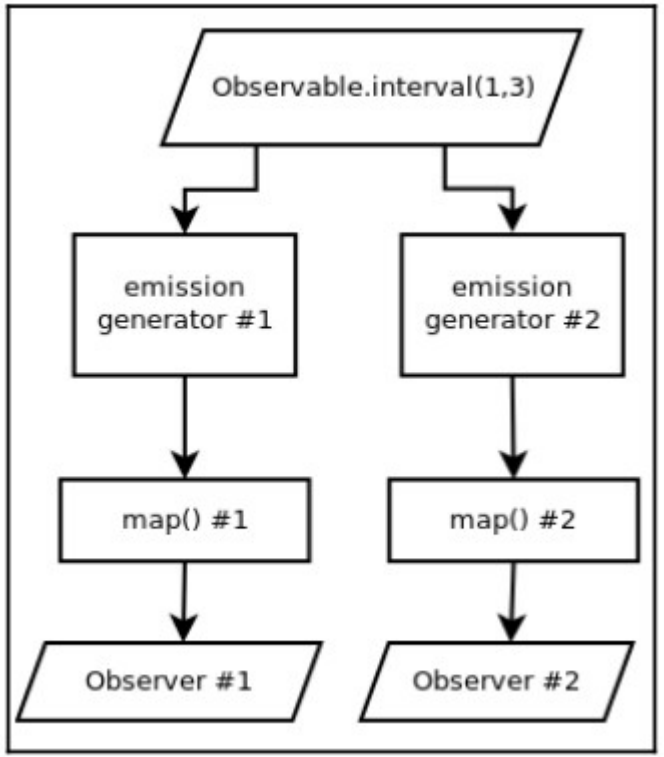
```
public static void main(String[] args) {
    Observable<Integer> ints = Observable.range( start: 1,  count: 3)
            .map(i -> randomInt());
    ints.subscribe(i -> System.out.println("Observer 1: " + i));
    ints.subscribe(i -> System.out.println("Observer 2: " + i));
}

public static int randomInt() { return ThreadLocalRandom.current().nextInt( bound: 100000);
```

```
Observer 1: 38895
Observer 1: 36858
Observer 1: 82955
Observer 2: 55957
Observer 2: 47394
Observer 2: 16996
```

MRC

- **multicastWithOperators:**
  - How multicasting works within a chain of operators, we are going to use Observable.range() and then map each emission to a random integer.



MRC

- **multicastWithOperators:**
  - Your first instinct might be to call publish() after Observable.range() to yield a ConnectableObservable.
    - Then, you may call the map() operator on it, followed by the subscribing and a connect() call

```
public static void main(String[] args) {
    ConnectableObservable<Integer> ints =
                                Observable.range(1, 3).publish();
    Observable<Integer> rInts = ints.map(i -> randomInt());
    rInts.subscribe(i -> System.out.println("Observer 1: " + i));
    rInts.subscribe(i -> System.out.println("Observer 2: " + i));
    ints.connect();
}
```

```
Observer 1: 99350
Observer 2: 96343
Observer 1: 4155
Observer 2: 75273
Observer 1: 14280
Observer 2: 97638
```

MRC

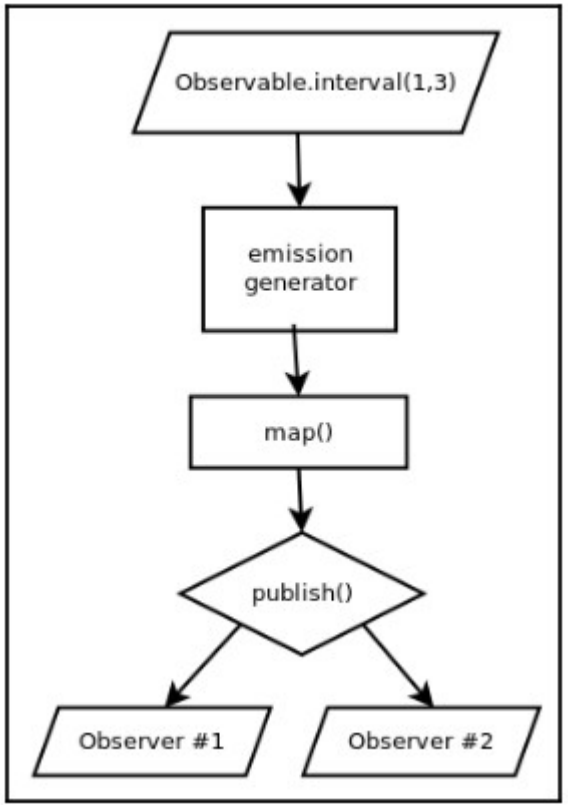
- **multicastWithOperators:**
  - To prevent the map() operator from yielding a separate stream for each Observer, we need to call publish() after map() instead

```
public static void main(String[] args) {
    ConnectableObservable<Integer> rInts =
          Observable.range(1, 3).map(i -> randomInt()).publish();
    rInts.subscribe(i -> System.out.println("Observer 1: " + i));
    rInts.subscribe(i -> System.out.println("Observer 2: " + i));
    rInts.connect();
}
```

```
Observer 1: 90125
Observer 2: 90125
Observer 1: 79156
Observer 2: 79156
Observer 1: 76782
Observer 2: 76782
```

MRC

- **multicastWithOperators:**
  - To prevent the map() operator from yielding a separate stream for each Observer, we need to call publish() after map() instead



MRC

- **When to multicast:**
  - Multicasting is helpful when you need to send the same data to several observers.
    - If the emitted data has to be **processed the same way** for each of these observers, do it before calling publish().
    - This prevents redundant work being done by multiple observers.
      - You may do this to i**ncrease performance, to reduce memory and CPU usage**, or simply because your **business logic requires** pushing the same emissions to all observers
    - Make cold observables multicast only when you are doing so for performance reasons and have multiple observers receiving the same data simultaneously

MRC

- **When to multicast:**
  - Remember that multicasting creates **hot ConnectableObservables**, and you have to be careful and time the connect() call so data is not missed by some observers
  - Typically in your API, keep your cold observables cold and call publish() when you need to make them hot
  - Do not multicast when there is only one Observer because multicasting can cause **an overhead**.
    - However, if there are multiple observers, you need to find the proxy point where you can multicast and consolidate the upstream operations.
    - This point is typically the boundary where observers have common operations upstream and diverge into different operations downstream.

MRC

- **multicastWithOperators:**
  - Best practice

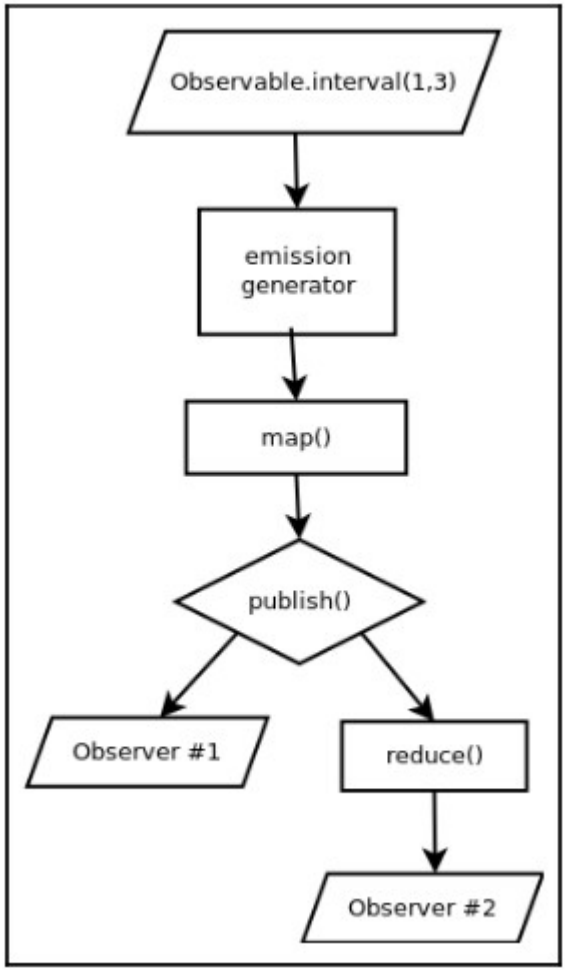
```
public static void main(String[] args) {
    ConnectableObservable<Integer> rInts =
          Observable.range(1, 3).map(i -> randomInt()).publish();
    //Observer 1 - print each random integer
    rInts.subscribe(i -> System.out.println("Observer 1: " + i));

    //Observer 2 - sum the random integers, then print
    rInts.reduce(0, (total, next) -> total + next)
         .subscribe(i -> System.out.println("Observer 2: " + i));
    rInts.connect();
}
```

```
Observer 1: 40021
Observer 1: 78962
Observer 1: 46146
Observer 2: 165129
```

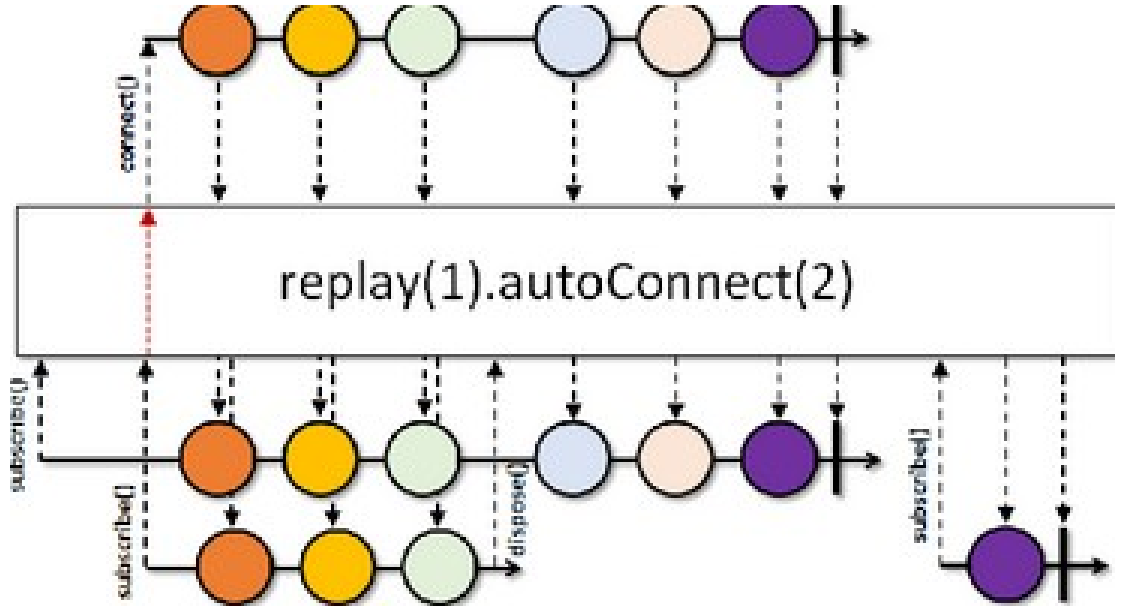
MRC

- **multicastWithOperators:**
  - Best practice



MRC

- **Multicast:autoconnect:**
  - Returns an Observable that automatically connects (at most once) to this ConnectableObservable when the specified number of Subscribers subscribe to it and calls the specified callback with the Subscription associated with the established connection



MRC

- **Multicast:autoconnect:**
  - There are definitely times you would want to manually call connect() on ConnectableObservable to precisely control when the emissions start firing. There are also operators that automatically call connect() for you, but with this convenience, it is important to have an awareness of their timing behavior.
    - Autoconnect(2): 2 observers specified

```
public static void main(String[] args) {
    Observable<Integer> rInts =
       Observable.range(1, 3).map(i -> randomInt()).publish()
                                                   .autoConnect(2);
    //Observer 1 - prints each random integer
    rInts.subscribe(i -> System.out.println("Observer 1: " + i));

    //Observer 2 - sums the random integers, then prints
     rInts.reduce(0, (total, next) -> total + next)
          .subscribe(i -> System.out.println("Observer 2: " + i));
}
```

```
Observer 1: 83428
Observer 1: 77336
Observer 1: 64970
Observer 2: 225734
```

MRC

- **Multicast:autoconnect:**
  - To demonstrate it, we can add a third Observer to our example but keep autoConnect(2) instead of autoConnect(3).
    - The third Observer is going to miss the emissions

```
Observer 1: 8198
Observer 1: 31718
Observer 1: 97915
Observer 2: 137831
```

```
public static void main(String[] args) {
    Observable<Integer> rInts =
       Observable.range(1, 3).map(i -> randomInt()).publish()
                                                   .autoConnect(2);
    //Observer 1 - prints each random integer
    rInts.subscribe(i -> System.out.println("Observer 1: " + i));
    //Observer 2 - sums the random integers, then print
    rInts.reduce(0, (total, next) -> total + next)
         .subscribe(i -> System.out.println("Observer 2: " + i));
    //Observer 3 - receives nothing
    rInts.subscribe(i -> System.out.println("Observer 3:" + i));
}
```

MRC

- **Multicast:autoconnect:**
  - If you want it to start firing on the first subscription and do not care about any subsequent observers missing previous emissions. Call publish() and autoConnect()

```
public static void main(String[] args) {
    Observable<Long> ints =
            Observable.interval( period: 1, TimeUnit.SECONDS) Observable<Long>
                    .publish() ConnectableObservable<Long>
                    .autoConnect();
    //Observer 1
    ints.subscribe(i -> System.out.println("Observer 1: " + i));
    sleep( millis: 3000);
    //Observer 2
    ints.subscribe(i -> System.out.println("Observer 2: " + i));
    sleep( millis: 3000);
}
```

```
Observer 1: 0
Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 2: 3
Observer 1: 4
Observer 2: 4
Observer 1: 5
Observer 2: 5
```

MRC

- **Multicast:refcount:**
  - The refCount() operator on ConnectableObservable is similar to **autoConnect(1)**, which fires after getting one subscription, with one important difference:
    - after all its observers have been disposed of, it disposes of itself and starts over **when a new subscription** comes in.
    - It does not persist the subscription to the source when it has no subscribers. When another subscription happens, it essentially **starts over**.

MRC

- **Multicast:refcount:**

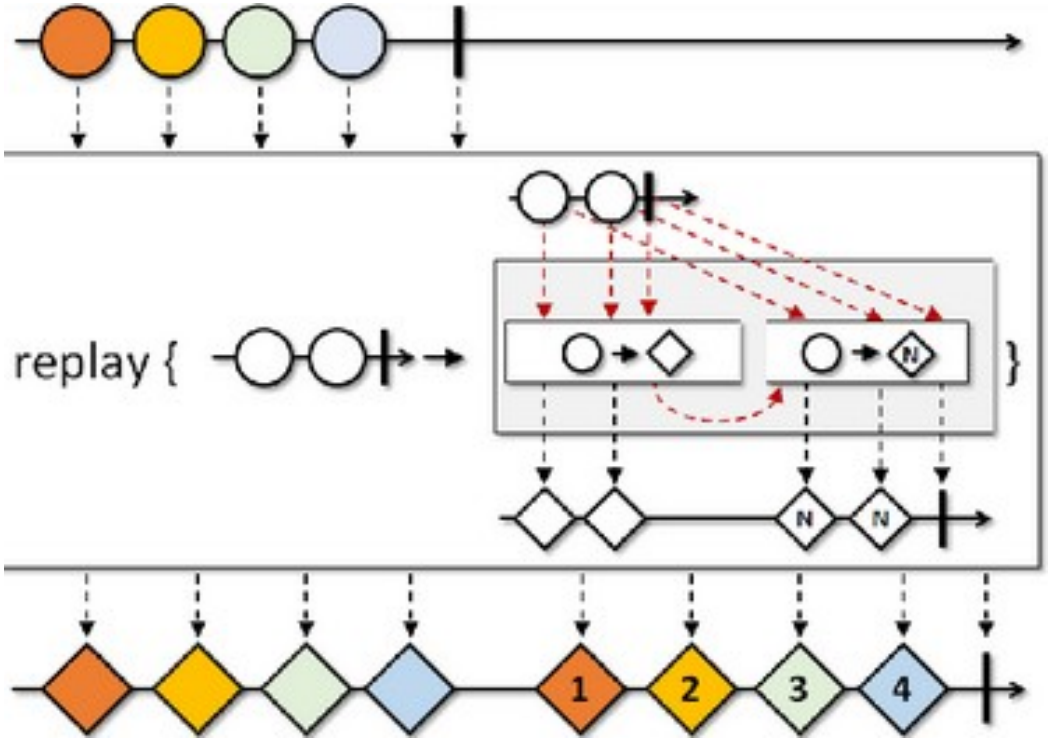
```
public static void main(String[] args) {
    Observable<Long> ints =
            Observable.interval(1, TimeUnit.SECONDS).publish()
                                                    .refCount();
    //Observer 1
    ints.take(5)
        .subscribe(l -> System.out.println("Observer 1: " + l));
    sleep(3000);
    //Observer 2
    ints.take(2)
        .subscribe(l -> System.out.println("Observer 2: " + l));
    sleep(3000);
    //There should be no more subscribers at this point

    //Observer 3
    ints.subscribe(l -> System.out.println("Observer 3: " + l));
    sleep(3000);
}
```

```
Observer 1: 0
Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 2: 3
Observer 1: 4
Observer 2: 4
Observer 3: 0
Observer 3: 1
Observer 3: 2
```

MRC

- **Replaying**
  - Returns an Observable that emits items that are the results of invoking a specified selector on the items emitted by a ConnectableObservable that shares a single subscription to the current Observable.



MRC

- **Replaying**
  - replay() with no arguments. This will replay all previous emissions to tardy observers and then emit current emissions as soon as the tardy Observer is caught up.
    - If we use Observable.interval() to emit every second

```
public static void main(String[] args) {
    Observable<Long> ints =
         Observable.interval(1, TimeUnit.SECONDS).replay()
                                                 .autoConnect();

    //Observer 1
    ints.subscribe(l -> System.out.println("Observer 1: " + l));
    sleep(3000);

    //Observer 2
    ints.subscribe(l -> System.out.println("Observer 2: " + l));
    sleep(3000);
}
```

```
Observer 1: 0
Observer 1: 1
Observer 1: 2
Observer 2: 0
Observer 2: 1
Observer 2: 2
Observer 1: 3
Observer 2: 3
Observer 1: 4
Observer 2: 4
Observer 1: 5
Observer 2: 5
```

MRC

- **Replaying**
  - If the source is infinite or you only care about the last few emissions, you might want to specify a buffer size by using an overloaded version of the replay() method, replay(int bufferSize)
    - If we call replay(2) on our second Observer to cache the last two emissions, it will not get 0, but it will receive 1 and 2.

```
Observer 1: 0
Observer 1: 1
Observer 1: 2
Observer 2: 1
Observer 2: 2
Observer 1: 3
Observer 2: 3
Observer 1: 4
Observer 2: 4
Observer 1: 5
Observer 2: 5
```

MRC

- **Replaying**
  - we use replay(1).autoConnect() to hold on to the last value. The second Observer receives only the last value, as expected

```
public static void main(String[] args) {
   Observable<String> src =
      Observable.just("Alpha", "Beta", "Gamma").replay(1)
                                               .autoConnect();
   //Observer 1
   src.subscribe(l -> System.out.println("Observer 1: " + l));

   //Observer 2
   src.subscribe(l -> System.out.println("Observer 2: " + l));
}
```

```
Observer 1: Alpha
Observer 1: Beta
Observer 1: Gamma
Observer 2: Gamma
```

MRC

- **Replaying**
  - use refCount() instead of autoConnect():
  - What happened here is that refCount() causes the cache (and the entire chain) to be disposed of and reset the moment Observer 1 is done, as there are no more observers.
    - When Observer 2 comes in, it starts all over and emits everything just like Observer 1 did, and another cache is built.

```
Observable<String> source = Observable.just("Alpha", "Beta", "Gamma")
                    .replay(1)
                    .refCount();
```

```
Observer 1: Alpha
Observer 1: Beta
Observer 1: Gamma
Observer 2: Alpha
Observer 2: Beta
Observer 2: Gamma
```

MRC

- **Replaying**
  - There are other overloads for replay(), particularly a version that allows specification of a time-based window.
    - In the following example, we construct an Observable.interval() that emits every 300 milliseconds and subscribe to it.

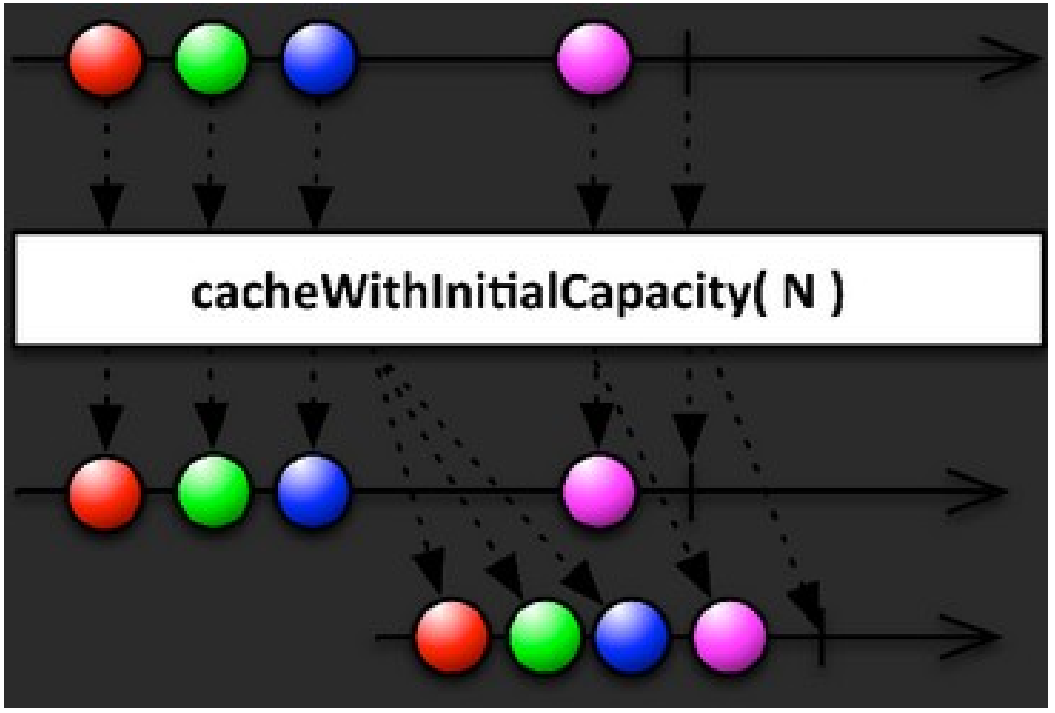
```
public static void main(String[] args) {
    Observable<Long> seconds =
       Observable.interval(300, TimeUnit.MILLISECONDS)
                 .map(l -> (l + 1) * 300) // map to elapsed millis
                 .replay(1, TimeUnit.SECONDS)
                 .autoConnect();
    //Observer 1
    seconds.subscribe(l -> System.out.println("Observer 1: " + l));
    sleep(2000);

    //Observer 2
    seconds.subscribe(l -> System.out.println("Observer 2: " + l));
    sleep(1000);
}
```

```
Observer 1: 300
Observer 1: 600
Observer 1: 900
Observer 1: 1200
Observer 1: 1500
Observer 1: 1800
Observer 2: 1500
Observer 2: 1800
Observer 1: 2100
Observer 2: 2100
Observer 1: 2400
```

MRC

- **Cache**
  - Returns an Observable that subscribes to the current Observable lazily, caches all of its events and replays them, in the same order as received, to all the downstream observers.



MRC

- **Cache**
  - When you want to cache all emissions indefinitely for the long term and do not need to control the subscription behavior to the source with ConnectableObservable, you can use the cache() operator.
    - You can also call cacheWithInitialCapacity()

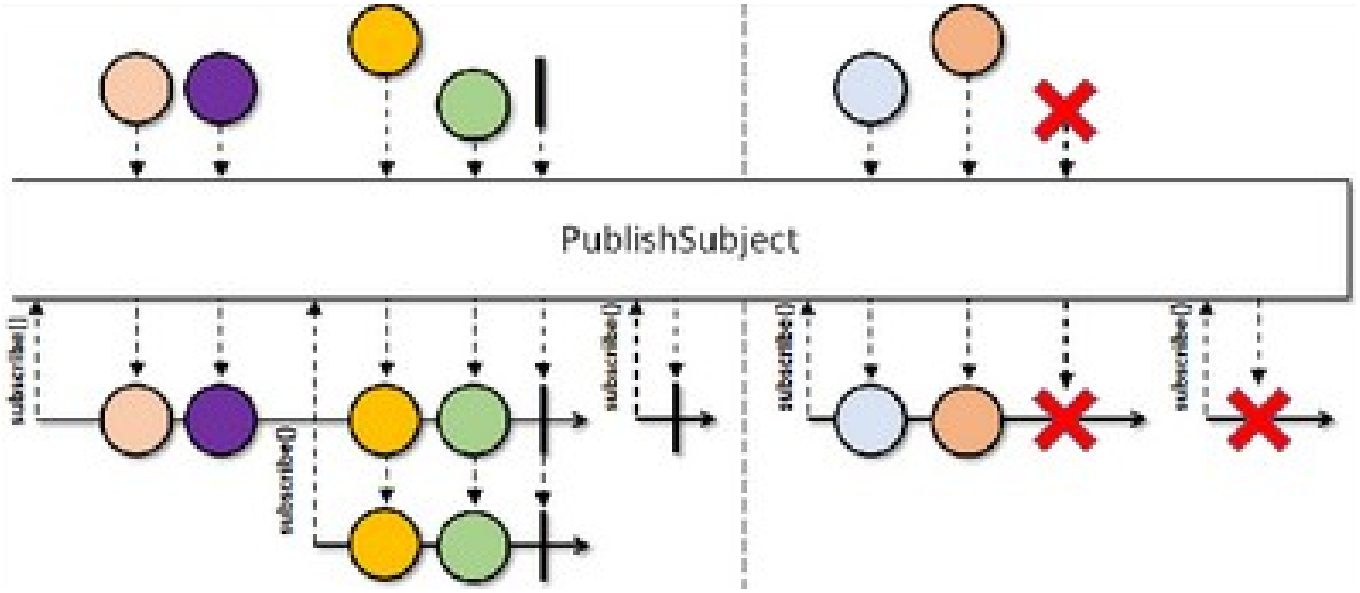
```
public static void main(String[] args) {
    Observable<Integer> cachedRollingTotals =
            Observable.just(6, 2, 5, 7, 1, 4, 9, 8, 3)
                    .scan(0, (total, next) -> total + next)
                    .cache();
    cachedRollingTotals.subscribe(System.out::println);
}
```

MRC

- **Subject**
  - Erik Meijer, the creator of ReactiveX, described them as the "mutable variables of reactive programming".
    - Just like a mutable variable is necessary at times even though you should strive for immutability, a Subject is sometimes a necessary tool to reconcile an imperative paradigm with a reactive one.
    - A Subject is both an Observer and an Observable, acting as a proxy multicasting device (similar to an event bus).
      - Used as last resort

MRC

- **Subject:PublishSubject**
  - A Subject that emits (multicasts) items to currently subscribed Observers and terminal events to current or late Observers.



MRC

- **Subject:PublishSubject**

```
public static void main(String[] args) {
    Subject<String> subject = PublishSubject.create();
    subject.map(String::length)
           .subscribe(System.out::println);

    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");
    subject.onComplete();
}
```

```
5
4
5
```

MRC

- **Subject:PublishSubject**
  - Use a Subject to eagerly subscribe to an unknown number of multiple source observables and consolidate their emissions in a single Observable object.

```
public static void main(String[] args) {
    Observable<String> source1 =
        Observable.interval(1, TimeUnit.SECONDS)
                  .map(l -> (l + 1) + " seconds");
    Observable<String> source2 =
        Observable.interval(300, TimeUnit.MILLISECONDS)
                  .map(l -> ((l + 1) * 300) + " milliseconds");
    Subject<String> subject = PublishSubject.create();
    subject.subscribe(System.out::println);
    source1.subscribe(subject);
    source2.subscribe(subject);
    sleep(3000);
}
```

```
300 milliseconds
600 milliseconds
900 milliseconds
1 seconds
1200 milliseconds
1500 milliseconds
1800 milliseconds
2 seconds
2100 milliseconds
2400 milliseconds
2700 milliseconds
3 seconds
3000 milliseconds
```

MRC

- **Subject:PublishSubject:when to use?**
  - We did not define the source emissions until the end—after all the observers were set up. With such a layout, the process no longer reads from left to right, and from top to bottom.
  - For example, if you move the onNext() calls as shown in the following example, you will not get any output because the Observer will miss these emissions:

```
public static void main(String[] args) {
    Subject<String> subject = PublishSubject.create();
    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");
    subject.onComplete();
    subject.map(String::length)
           .subscribe(System.out::println);
}
```

MRC

- **Subject:Serialize**
  - A critical gotcha to note with Subjects is this: the onSubscribe(), onNext(), onError(), and onComplete() calls are not threadsafe!
    - If you have multiple threads calling these four methods, emissions could start to overlap and break the Observable contract, which demands that emissions happen sequentially.
    - If this happens, a good practice to adopt is to call toSerialized() on the Subject
    - This will safely sequentialize concurrent event calls so that no train wreck occurs downstream.

```
Subject<String> subject =
PublishSubject.<String>create().toSerialized();
```

MRC

- **Subject:BehaviourSubject**
  - It behaves almost the same way as PublishSubject, but it also replays the last emitted item to each new Observer downstream.
    - This is somewhat like putting replay(1).autoConnect()

```
public static void main(String[] args) {
    Subject<String> subject = BehaviorSubject.create();
    subject.subscribe(s -> System.out.println("Observer 1: " + s));
    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");
    subject.subscribe(s -> System.out.println("Observer 2: " + s));
}
```

```
Observer 1: Alpha
Observer 1: Beta
Observer 1: Gamma
Observer 2: Gamma
```

MRC

- **Subject:ReplaySubject**
  - The ReplaySubject class behaves similar to PublishSubject followed by a cache() operator.
    - It immediately captures emissions regardless of the presence of a downstream Observer and optimizes the caching to occur inside the Subject itself.

```
public static void main(String[] args) {
    Subject<String> subject = ReplaySubject.create();
    subject.subscribe(s -> System.out.println("Observer 1: " + s));
    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");
    subject.onComplete();
    subject.subscribe(s -> System.out.println("Observer 2: " + s)
}
```

```
Observer 1: Alpha
Observer 1: Beta
Observer 1: Gamma
Observer 2: Alpha
Observer 2: Beta
Observer 2: Gamma
```

MRC

- **Subject:AsyncSubject**
  - The AsyncSubject class has a highly tailored, finite-specific behavior: it pushes only the last value it receives, followed by an onComplete() event.

```
public static void main(String[] args) {
    Subject<String> subject = AsyncSubject.create();
    subject.subscribe(s -> System.out.println("Observer 1: " + s),
                      Throwable::printStackTrace,
                      () -> System.out.println("Observer 1 done!")
    );
    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");
    subject.onComplete();
    subject.subscribe(s -> System.out.println("Observer 2: " + s),
                      Throwable::printStackTrace,
                      () -> System.out.println("Observer 2 done!")
    );
}
```

```
Observer 1: Gamma
Observer 1 done!
Observer 2: Gamma
Observer 2 done!
```

MRC

- **Subject:UnicastSubject**
  - it buffers all the emissions it receives until an Observer subscribes to it, and then it releases all the emissions to the Observer and clears its cache.

```
public static void main(String[] args) {
    Subject<String> subject = UnicastSubject.create();
    Observable.interval(300, TimeUnit.MILLISECONDS)
              .map(l -> ((l + 1) * 300) + " milliseconds")
              .subscribe(subject);
    sleep(2000);
    subject.subscribe(s -> System.out.println("Observer 1: " + s));
    sleep(2000);
}
```

```
Observer 1: 300 milliseconds
Observer 1: 600 milliseconds
Observer 1: 900 milliseconds
Observer 1: 1200 milliseconds
Observer 1: 1500 milliseconds
Observer 1: 1800 milliseconds
Observer 1: 2100 milliseconds
Observer 1: 2400 milliseconds
Observer 1: 2700 milliseconds
Observer 1: 3000 milliseconds
Observer 1: 3300 milliseconds
```

Agenda

- Introduction
- Observables and Observer
- Operators
- Combining Observables
- Multicasting, Replaying and Caching
- **Concurrency**
- Switching, Throttling, Windowing and buffering
- Flowables and Backpressure

Concurrency

- **Introduction**
  - Note how both observables fire emissions as each one is slowed by 0-3 seconds in the map() operation.
    - More importantly, note how the first Observable firing Alpha, Beta, and Gamma must finish first and call onComplete() before firing the second Observable emitting the numbers 1 through 3.

```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
              .map(s -> intenseCalculation((s)))
              .subscribe(System.out::println);
    Observable.range(1, 3)
              .map(s -> intenseCalculation((s)))
              .subscribe(System.out::println);
}
```

```
Alpha
Beta
Gamma
1
2
3
```

Concurrency

- **subscribeOn**
  - Using the subscribeOn() operator we fire both observables at the same time rather than waiting for one to complete before starting the other.

```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma")
              .subscribeOn(Schedulers.computation())
              .map(s -> intenseCalculation((s)))
              .subscribe(System.out::println);
    Observable.range(1, 3)
              .subscribeOn(Schedulers.computation())
              .map(s -> intenseCalculation((s)))
              .subscribe(System.out::println);
    sleep(20000);
}
```

```
1
Alpha
2
3
Beta
Gamma
```

Concurrency

- **SubscribeOn-Zip/Merge/etc**
  - RxJava operators such as merge() and zip() can work with observables on different threads safely.

```
public static void main(String[] args) {
    Observable<String> source1 =
            Observable.just("Alpha", "Beta", "Gamma")
                    .subscribeOn(Schedulers.computation())
                    .map(s -> intenseCalculation((s)));
    Observable<Integer> source2 =
            Observable.range(1, 3)
                    .subscribeOn(Schedulers.computation())
                     .map(s -> intenseCalculation((s)));
    Observable.zip(source1, source2, (s, i) -> s + "-" + i)
            .subscribe(System.out::println);
    sleep(20000);
}
```

```
Alpha-1
Beta-2
Gamma-3
```

Concurrency

- **BlockingSubscribe():**
  - Can be used in place of subscribe() to stop and wait for onComplete() to be called before the main thread is allowed to proceed and exit the application

```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
              .subscribeOn(Schedulers.computation())
              .map(Ch6_6::intenseCalculation)
              .blockingSubscribe(System.out::println,
                                 Throwable::printStackTrace,
                                 () -> System.out.println("Done!"));
}
```

```
Alpha
Beta
Gamma
Delta
Epsilon
Done!
```

Concurrency

- **Schedulers**
  - In Java, ExecutorService is used as a thread pool.
    - However, RxJava implements its own concurrency abstraction called Scheduler.
      - This defines methods and rules that an actual concurrency provider such as an ExecutorService or actor system must obey.
      - Many of the default Scheduler implementations can be found in the Schedulers static factory class.
      - For a given Observer, a Scheduler provides a thread from a pool that will push the emissions from the Observable.
      - When onComplete() is called, the operation will be disposed of and the thread will be given back to the pool

Concurrency

- **Schedulers:Types:Computation**
  - Computation Scheduler is created by the factory Schedulers.computation() method
  - Computation Scheduler that is created by the factory Schedulers.computation() method
  - Computational tasks (such as math, algorithms, and complex logic) may utilize cores to their fullest extent.

```
Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
          .subscribeOn(Schedulers.computation());
```

Concurrency

- **Schedulers:Types:Computation**
  - A number of operators and factories will use the computation Scheduler by default unless you specify a different one as an argument.
  - These include one or more overloads for interval(), delay(), timer(), timeout(), buffer(), take(), skip(), takeWhile(), skipWhile(), window(), and a few others.

Concurrency

- **Schedulers:Types:IO**
  - I/O tasks, such as reading from and writing to databases, web requests, and disk storage use little CPU power and often have idle time waiting for the data to be sent or received.
  - This allows threads to be created more liberally, and Schedulers.io() is appropriate for this.
  - It maintains as many threads as there are tasks and dynamically grows the number of threads, caches them, and discards the threads when they are not needed.

```
Database db = Database.from(conn);
Observable<String> customerNames =
      db.select("select name from customer")
        .getAs(String.class)
        .subscribeOn(Schedulers.io());
```

Concurrency

- **Schedulers:Types:IO**
  - I/O tasks, such as reading from and writing to databases, web requests, and disk storage use little CPU power and often have idle time waiting for the data to be sent or received.
  - This allows threads to be created more liberally, and Schedulers.io() is appropriate for this.
  - It maintains as many threads as there are tasks and dynamically grows the number of threads, caches them, and discards the threads when they are not needed.

Concurrency

- **Schedulers:Types:NewThread**
  - The Schedulers.newThread() factory returns a Scheduler that does not pool threads at all.
  - It creates a new thread for each Observer and then destroys the thread when it is not needed anymore.
  - This may be helpful in cases where you want to create, use, and then destroy a thread immediately so that it does not take up memory.

```
Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
          .subscribeOn(Schedulers.newThread());
```

Concurrency

- **Schedulers:Types:Single**
  - When you want to run tasks sequentially on a single thread, you can use Schedulers.single() to create a Scheduler.
  - This is backed by a single-threaded implementation appropriate for event looping.
  - It can also be helpful to isolate fragile, non-thread-safe operations to a single thread.

```
Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
          .subscribeOn(Schedulers.single());
```

Concurrency

- **Schedulers:Types:Trampoline**
  - A Scheduler created by Schedulers.trampoline() is an interesting one. Used primarily in RxJava's internal implementation.

Concurrency

- **Schedulers:Types:ExecutorService**
  - It is also possible to build a Scheduler from a standard Java ExecutorService pool. Usually to have more custom and fine-tuned control over your thread management policies based on the load factor.
  - Wrap it inside a Scheduler implementation by calling Schedulers.from()

```
public static void main(String[] args) {
    int numberOfThreads = 20;
    ExecutorService executor =
            Executors.newFixedThreadPool(numberOfThreads);
    Scheduler scheduler = Schedulers.from(executor);
    Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
              .subscribeOn(scheduler)
              .doFinally(executor::shutdown)
              .subscribe(System.out::println);
}
```

Concurrency

- **Schedulers:Startup and Shutdown**
  - Each default Scheduler is lazily instantiated.
    - The Scheduler created by the computation(), io(), newThread(), single(), or trampoline() factory method can be disposed of at any time by calling its shutdown() method.
    - Alternatively, all created schedulers can be disposed of by calling Schedulers.shutdown()

Concurrency

- **subscribeOn**
  - The subscribeOn() operator suggests to the source Observable which Scheduler to use and how to execute operations on one of its threads.
    - If that source is not already tied to a particular Scheduler, it will use the specified Scheduler.
    - It will then push emissions all the way to the final Observer using that thread.
    - You can put subscribeOn() **anywhere in the Observable chain**, and it will suggest to the source Observable which thread to execute emissions with.

Concurrency

- **subscribeOn**
  - The subscribeOn() operator communicates upstream to Observable.just() which Scheduler to use no matter where you put it.
    - For clarity, though, you should place it as close to the source as possible, as in the following code:

```
Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
          .subscribeOn(Schedulers.computation()) //preferred
          .map(String::length)
          .filter(i -> i > 5)
          .subscribe(System.out::println);

Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
          .map(String::length)
          .filter(i -> i > 5)
          .subscribeOn(Schedulers.computation())
          .subscribe(System.out::println);
```

Concurrency

- **subscribeOn**
  - Having multiple observers to the same Observable with subscribeOn() results in each one getting its own thread (or have them waiting for an available thread if none are available).

```
public static void main(String[] args) {
    Observable<Integer> lengths =
        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
                .subscribeOn(Schedulers.computation())
                .map(SubscribeOn6::intenseCalculation)
                .map(String::length) Observable<Integer>
                .publish() ConnectableObservable<Integer>
                .autoConnect( numberOfSubscribers: 2);
    lengths.subscribe(i -> System.out.println("Received " + i +
            " on thread " + Thread.currentThread().getName()));
    lengths.subscribe(i -> System.out.println("Received " + i +
            " on thread " + Thread.currentThread().getName()));
    sleep( millis: 10000);
}
```

Concurrency

- **subscribeOn**
  - In the Observer, you can print the executing thread's name by calling Thread.currentThread().getName()

```
Received 5 on thread RxComputationThreadPool-2
Received 4 on thread RxComputationThreadPool-2
Received 5 on thread RxComputationThreadPool-2
Received 5 on thread RxComputationThreadPool-2
Received 5 on thread RxComputationThreadPool-1
Received 7 on thread RxComputationThreadPool-2
Received 4 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
```

Concurrency

- **SubscribeOn**
  - Multicasting will use a single thread for both observers
  - subscribeOn() must be placed before the multicast operators

```
public static void main(String[] args) {
   Observable<Integer> lengths =
      Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
                .subscribeOn(Schedulers.computation())
                .map(Ch6_9::intenseCalculation)
                .map(String::length)
                .publish()
                .autoConnect(2);
      lengths.subscribe(i -> System.out.println("Received " + i +
                " on thread " + Thread.currentThread().getName()));
      lengths.subscribe(i -> System.out.println("Received " + i +
                " on thread " + Thread.currentThread().getName()));
      sleep(10000);
  }
```

Concurrency

- **SubscribeOn**
  - Multicasting will use a single thread for both observers
  - subscribeOn() must be placed before the multicast operators

```
Received 5 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
Received 4 on thread RxComputationThreadPool-1
Received 4 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
```

Concurrency

- **SubscribeOn**
  - Most Observable factories, such as Observable.fromIterable() and Observable.just(), create an Observable that emits items on the Scheduler specified by subscribeOn().
  - For factories such as Observable.fromCallable() and Observable.defer(), the initialization of these sources also runs on the specified Scheduler when using subscribeOn()

Concurrency

- **SubscribeOn**
  - Use Observable.fromCallable() to wait on a URL response, make it work on the I/O Scheduler so that the main thread is not blocking and waiting for it

```
public static void main(String[] args) {
    String href = "https://api.github.com/users/slowbreathing";
    Observable.fromCallable(() -> getResponse(href))
            .subscribeOn(Schedulers.io())
            .subscribe(System.out::println);
    sleep( millis: 10000);
}

1 usage
private static String getResponse(String path) {
    try {
        return new Scanner(new URL(path).openStream(),
                s: "UTF-8").useDelimiter( s: "\\A").next();
    } catch (Exception e) {
        return e.getMessage();
    }
}
```

Concurrency

- **SubscribeOn**
  - subscribeOn() operator has no practical effect with certain sources (and keeps a worker thread unnecessarily on standby until that operation terminates).
    - This might be because an Observable already uses a Scheduler.

```
public static void main(String[] args) {
    Observable.interval( period: 1, TimeUnit.SECONDS)
              .subscribeOn(Schedulers.newThread())
              .subscribe(i -> System.out.println("Received " + i +
                    " on thread " + Thread.currentThread().getName()));
    sleep( millis: 5000);
}
```

```
Received 0 on thread RxComputationThreadPool-1
Received 1 on thread RxComputationThreadPool-1
Received 2 on thread RxComputationThreadPool-1
Received 3 on thread RxComputationThreadPool-1
Received 4 on thread RxComputationThreadPool-1
```

Concurrency

- **SubscribeOn**
  - Provide a Scheduler as a third argument to specify a different Scheduler to use

```
public static void main(String[] args) {
    Observable.interval( period: 1, TimeUnit.SECONDS, Schedulers.newThread())
              .subscribe(i -> System.out.println("Received " + i +
                                 " on thread " + Thread.currentThread().getName()));
    sleep( millis: 5000);
}
```

```
Received 0 on thread RxNewThreadScheduler-1
Received 1 on thread RxNewThreadScheduler-1
Received 2 on thread RxNewThreadScheduler-1
Received 3 on thread RxNewThreadScheduler-1
Received 4 on thread RxNewThreadScheduler-1
```

Concurrency

- **SubscribeOn**
  - This brings up the following point: if you have multiple subscribeOn() calls on a given Observable chain, the top-most one, or the one closest to the source, will win and cause any subsequent ones to have no practical effect (other than unnecessary resource usage).
  - If subscribeOn() is used with Schedulers.computation() and downstream, another subscribeOn() with Schedulers.io()

Concurrency

- **SubscribeOn**

```
public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
            .subscribeOn(Schedulers.computation())
            .filter(s -> s.length() == 5)
            .subscribeOn(Schedulers.io())
            .subscribe(i -> System.out.println("Received " + i +
               " on thread " + Thread.currentThread().getName()));
    sleep(5000);
}
```

```
Received Alpha on thread RxComputationThreadPool-1
Received Gamma on thread RxComputationThreadPool-1
Received Delta on thread RxComputationThreadPool-1
```

Concurrency

- **ObserveOn**
  - The subscribeOn() operator instructs the source Observable which Scheduler to emit emissions on.
    - If subscribeOn() is the only concurrent operation in an Observable chain, the thread from that Scheduler will work the entire Observable chain, pushing emissions from the source all the way to the final Observer.
    - The observeOn() operator, however, will intercept emissions at that point in the Observable chain and switch them to a different Scheduler going forward.
    - Unlike subscribeOn(), the **placement of observeOn() matters**. It leaves all operations upstream on the default or subscribeOn()-defined Scheduler, but switches to a different Scheduler downstream.

Concurrency

- **ObserveOn**

```
public static void main(String[] args) {
   Observable.just("WHISKEY/27653/TANGO",
                           "6555/BRAVO","232352/5675675/FOXTROT")
         .subscribeOn(Schedulers.io())    //Starts on I/O scheduler
         .flatMap(s -> Observable.fromArray(s.split("/")))
         .observeOn(Schedulers.computation()) //Switches to
                                              // computation scheduler
         .filter(s -> s.matches("[0-9]+"))
         .map(Integer::valueOf)
         .reduce((total, next) -> total + next)
         .subscribe(i -> System.out.println("Received " + i +
               " on thread " + Thread.currentThread().getName()));
   sleep(1000);
}
```

```
Received 5942235 on thread RxComputationThreadPool-1
```

Concurrency

- **ObserveOn**
  - ObserveOn() can be used multiple times to switch Schedulers more than once. Write computed sum to a disk and write it in a file.

```
Observable.just("WHISKEY/27653/TANGO", "6555/BRAVO",
                                       "232352/5675675/FOXTROT") Observable<String>
    .subscribeOn(Schedulers.io())
    .flatMap(s -> Observable.fromArray(s.split( s: "/")))
    .doOnNext(s -> System.out.println("Split out " + s +
            " on thread " + Thread.currentThread().getName()))
    //Happens on Computation Scheduler
    .observeOn(Schedulers.computation())
    .filter(s -> s.matches( s: "[0-9]+"))
    .map(Integer::valueOf) Observable<Integer>
    .reduce((total, next) -> total + next) Maybe<Integer>
    .doOnSuccess(i -> System.out.println("Calculated sum" + i +
            " on thread " + Thread.currentThread().getName()))
    //Switch back to IO Scheduler
    .observeOn(Schedulers.io())
    .map(i -> i.toString()) Maybe<String>
    .doOnSuccess(s -> System.out.println("Writing " + s +
       " to file on thread " + Thread.currentThread().getName()))
    .subscribe(s -> write(s, Paths.get(home , ...strings: "sample.txt").toString()));
```

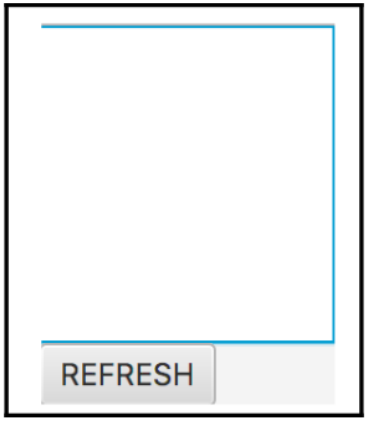
Concurrency

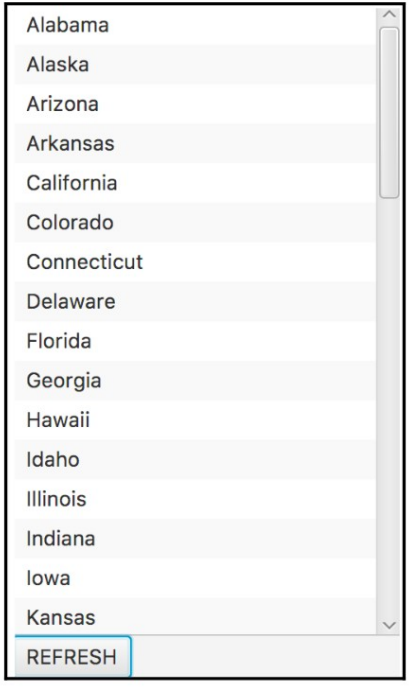
- **ObserveOn**
  - ObserveOn() can be used multiple times to switch Schedulers more than once. Write computed sum to a disk and write it in a file.

```
Split out WHISKEY on thread RxCachedThreadScheduler-1
Split out 27653 on thread RxCachedThreadScheduler-1
Split out TANGO on thread RxCachedThreadScheduler-1
Split out 6555 on thread RxCachedThreadScheduler-1
Split out BRAVO on thread RxCachedThreadScheduler-1
Split out 232352 on thread RxCachedThreadScheduler-1
Split out 5675675 on thread RxCachedThreadScheduler-1
Split out FOXTROT on thread RxCachedThreadScheduler-1
Calculated sum 5942235 on thread RxComputationThreadPool-1
Writing 5942235 to file on thread RxCachedThreadSchedule
```

Concurrency

- **ObserveOn**
  - Clicking the REFRESH button will emit an event but switch it to an I/O Scheduler where the work is done to retrieve the U.S. states. When the response is ready, it emits a List<String> and puts it back on the JavaFX Scheduler to be displayed in a ListView.





Concurrency

- **Parallelization**
  - This process can take a while due to the sleep()

```
public static void main(String[] args) {
    Observable.range(1, 10)
            .map(i -> intenseCalculation(i))
            .subscribe(i -> System.out.println("Received " + i +
                                               " " + LocalTime.now()));
}
public static <T> T intenseCalculation(T value) {
    sleep(ThreadLocalRandom.current().nextInt(3000));
    return value;
}
```

```
Received 1 19:11:41.812
Received 2 19:11:44.174
Received 3 19:11:45.588
Received 4 19:11:46.034
Received 5 19:11:47.059
Received 6 19:11:49.569
Received 7 19:11:51.259
Received 8 19:11:54.192
Received 9 19:11:56.196
Received 10 19:11:58.926
```

Concurrency

- **Parallelization**
  - let's wrap each emission into Observable.just(), use subscribeOn() to emit it on Schedulers.computation()

```
public static void main(String[] args) {
    Observable.range(1, 10)
              .flatMap(i -> Observable.just(i)
                              .subscribeOn(Schedulers.computation())
                              .map(i2 -> intenseCalculation(i2)))
             .subscribe(i -> System.out.println("Received " + i +
                             " " + LocalTime.now() + " on thread " +
                                 Thread.currentThread().getName()));
    sleep(20000);
}
```

```
Received 1 19:28:11.163 on thread RxComputationThreadPool-1
Received 7 19:28:11.381 on thread RxComputationThreadPool-7
Received 9 19:28:11.534 on thread RxComputationThreadPool-1
Received 6 19:28:11.603 on thread RxComputationThreadPool-6
Received 8 19:28:11.629 on thread RxComputationThreadPool-8
Received 3 19:28:12.214 on thread RxComputationThreadPool-3
Received 4 19:28:12.961 on thread RxComputationThreadPool-4
Received 5 19:28:13.274 on thread RxComputationThreadPool-5
Received 2 19:28:13.374 on thread RxComputationThreadPool-2
Received 10 19:28:14.335 on thread RxComputationThreadPool-2
```

Concurrency

- **Parallelization**
  - let's wrap each emission into Observable.just(), use subscribeOn() to emit it on Schedulers.computation()

```
public static void main(String[] args) {
    int coreCount = Runtime.getRuntime().availableProcessors();
    AtomicInteger assigner = new AtomicInteger(0);
    Observable.range(1, 10)
            .groupBy(i -> assigner.incrementAndGet() % coreCount)
            .flatMap(grp -> grp.observeOn(Schedulers.io())
                               .map(i2 -> intenseCalculation(i2)))
            .subscribe(i -> System.out.println("Received " + i +
                          " " + LocalTime.now() + " on thread " +
                           Thread.currentThread().getName()));
    sleep(20000);
}
```

```
Received 8 20:27:03.291 on thread RxCachedThreadScheduler-8
Received 6 20:27:03.446 on thread RxCachedThreadScheduler-6
Received 5 20:27:03.495 on thread RxCachedThreadScheduler-5
Received 4 20:27:03.681 on thread RxCachedThreadScheduler-4
Received 7 20:27:03.989 on thread RxCachedThreadScheduler-7
Received 2 20:27:04.797 on thread RxCachedThreadScheduler-2
Received 1 20:27:05.172 on thread RxCachedThreadScheduler-1
Received 9 20:27:05.327 on thread RxCachedThreadScheduler-1
Received 10 20:27:05.913 on thread RxCachedThreadScheduler-2
Received 3 20:27:05.957 on thread RxCachedThreadScheduler-3
```

Concurrency

- **unsubscribeOn**
  - Disposing of an Observable can be an expensive (in terms of the time it takes) operation, depending on the nature of the source. For instance, if the Observable emits the results of a database query using RxJava-JDBC, (https://github.com/davidmoten/rxjava-jdbc)

```
public static void main(String[] args) {
    Disposable d = Observable.interval(1, TimeUnit.SECONDS)
      .doOnDispose(() -> System.out.println("Disposing on thread "
                                + Thread.currentThread().getName()))
            .subscribe(i -> System.out.println("Received " + i));
    sleep(3000);
    d.dispose();
    sleep(3000);
}}
```

```
Received 0
Received 1
Received 2
Disposing on thread main
```

Concurrency

- **unsubscribeOn**
  - Add unsubscribeOn() and set it to be executed on the Schedulers.io() scheduler.
  - unsubscribeOn() affects all the operations upstream:

```
public static void main(String[] args) {
    Disposable d = Observable.interval(1, TimeUnit.SECONDS)
       .doOnDispose(() -> System.out.println("Disposing on thread "
                                + Thread.currentThread().getName()))
       .unsubscribeOn(Schedulers.io())
       .subscribe(i -> System.out.println("Received " + i));
    sleep(3000);
    d.dispose();
    sleep(3000);
}
```

```
Received 0
Received 1
Received 2
Disposing on thread RxCachedThreadScheduler-1
```

Agenda

- Introduction
- Observables and Observer
- Operators
- Combining Observables
- Multicasting, Replaying and Caching
- Concurrency
- **Switching, Throttling, Windowing and buffering**
- Flowables and Backpressure

STWB

- **Buffer**
  - The simplest overload for buffer() accepts a count argument as the buffer size and groups emissions in the batches of the specified size. To batch up emissions into lists of eight elements.

```
public static void main(String[] args) {
    Observable.range(1, 50)
              .buffer(8)
              .subscribe(System.out::println);
}
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[9, 10, 11, 12, 13, 14, 15, 16]
[17, 18, 19, 20, 21, 22, 23, 24]
[25, 26, 27, 28, 29, 30, 31, 32]
[33, 34, 35, 36, 37, 38, 39, 40]
[41, 42, 43, 44, 45, 46, 47, 48]
[49, 50]
```

STWB

- **Buffer**
  - Supply another argument in the buffer() overload, which is a bufferSupplier lambda expression that puts emissions in another collection, such as HashSet

```
public static void main(String[] args) {
    Observable.range(1,50)
              .buffer(8, HashSet::new)
              .subscribe(System.out::println);
}
```

STWB

- **Buffer**
  - You can also provide a skip argument that specifies how many items should be skipped before starting a new buffer. If skip is equal to count, the skip has no effect. But if they are different, you can get some interesting behavior.

```
public static void main(String[] args) {
    Observable.range(1, 10)
              .buffer(2, 3)
              .subscribe(System.out::println);
}
```

```
[1, 2]
[4, 5]
[7, 8]
[10]
```

STWB

- **Buffer**
  - If the skip value is smaller than count

```
public static void main(String[] args) {
    Observable.range(1, 10)
              .buffer(3, 1)
              .subscribe(System.out::println);
}
```

```
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
[4, 5, 6]
[5, 6, 7]
[6, 7, 8]
[7, 8, 9]
[8, 9, 10]
[9, 10]
[10]
```

STWB

- **Buffer**
  - Use buffer(2,1) to emit the previous emission and the next emission together

```
public static void main(String[] args) {
    Observable.range(1, 10)
              .buffer(2, 1)
              .filter(c -> c.size() == 2)
              .subscribe(System.out::println);
}
```

```
[1, 2]
[2, 3]
[3, 4]
[4, 5]
[5, 6]
[6, 7]
[7, 8]
[8, 9]
[9, 10]
```

STWB

- **Buffer:Time based**
  - You can use the buffer(long timespan, TimeUnit unit) operator at fixed time intervals by providing timespan and unit values

```
public static void main(String[] args) {
    Observable.interval(300, TimeUnit.MILLISECONDS)
              .map(i -> (i + 1) * 300)
              .buffer(1, TimeUnit.SECONDS)
              .subscribe(System.out::println);
    sleep(4000);
}
```

```
[300, 600, 900]
[1200, 1500, 1800]
[2100, 2400, 2700]
[3000, 3300, 3600, 3900]
```

STWB

- **Buffer:Time based**
  - Leverage a third count argument to provide a maximum buffer size. This will result in a buffer emission at each time interval or when count is reached, whichever happens first.

```
public static void main(String[] args) {
    Observable.interval(300, TimeUnit.MILLISECONDS)
              .map(i -> (i + 1) * 300)
              .buffer(1, TimeUnit.SECONDS, 2)
              .subscribe(System.out::println);
    sleep(5000);
}
```

```
[300, 600]
[900]
[1200, 1500]
[1800]
[2100, 2400]
[2700]
[3000, 3300]
[3600, 3900]
[]
[4200, 4500]
[4800]
```

STWB

- **Buffer:Boundary based**
  - 300-millisecond emissions buffered every 1 second using this technique.

```
public static void main(String[] args) {
  Observable<Long> cutOffs =
                          Observable.interval(1, TimeUnit.SECONDS);
  Observable.interval(300, TimeUnit.MILLISECONDS)
            .map(i -> (i + 1) * 300)
            .buffer(cutOffs)
            .subscribe(System.out::println);
  sleep(5000);
}
```

```
[300, 600, 900]
[1200, 1500, 1800]
[2100, 2400, 2700]
[3000, 3300, 3600, 3900]
[4200, 4500, 4800]
```

STWB

- **Window**
  - The window() operator is almost identical to the buffer() operator, except that it buffers into another Observable rather than a collection.
  - This results in an Observable<Observable<T>> that emits observables.
  - This allows emissions to be worked with immediately as they become available, rather than waiting for each list or collection to be finalized and emitted.
  - The window() operator is also convenient to work with if you want to use operators to transform each batch.

STWB

- **Window:Fixed**
  - Concatenate emissions into strings using a pipe (|), as a separator

```
public static void main(String[] args) {
    Observable.range(1, 50)
            .window(8)
            .flatMapSingle(obs -> obs.reduce("", (total, next) ->
                    total + (total.equals("") ? "" : "|") + next))
            .subscribe(System.out::println);
}
```

```
1|2|3|4|5|6|7|8
9|10|11|12|13|14|15|16
17|18|19|20|21|22|23|24
25|26|27|28|29|30|31|32
33|34|35|36|37|38|39|40
41|42|43|44|45|46|47|48
49|50
```

STWB

- **Window:Fixed**
  - Window size is 2, but skip three items

```
public static void main(String[] args) {
    Observable.range(1, 50)
            .window(2, 3)
            .flatMapSingle(obs -> obs.reduce("", (total, next) ->
                    total + (total.equals("") ? "" : "|") + next))
            .subscribe(System.out::println);
}
```

```
1|2
4|5
7|8
10|11
13|14
16|17
19|20
22|23
25|26
28|29
31|32
34|35
37|38
40|41
43|44
46|47
49|50
```

STWB

- **Window:time based**
  - Create an Observable that emits every 300 milliseconds and then slice it into separate observables every 1 second.

```
public static void main(String[] args) {
    Observable.interval(300, TimeUnit.MILLISECONDS)
            .map(i -> (i + 1) * 300)
            .window(1, TimeUnit.SECONDS)
            .flatMapSingle(obs -> obs.reduce("", (total, next) ->
                    total + (total.equals("") ? "" : "|") + next))
            .subscribe(System.out::println);
    sleep(5000);
}
```

```
300|600|900
1200|1500|1800
2100|2400|2700
3000|3300|3600|3900
4200|4500|4800
```

STWB

- **Window:boundary based**
  - the window() operator is similar to buffer(), so it should not come as a surprise that it is possible to use another Observable as a boundary value in the window() operator.

```
public static void main(String[] args) {
   Observable<Long> cutOffs =
                           Observable.interval(1, TimeUnit.SECONDS);
   Observable.interval(300, TimeUnit.MILLISECONDS)
             .map(i -> (i + 1) * 300)
             .window(cutOffs)
             .flatMapSingle(obs -> obs.reduce("", (total, next) ->
                     total + (total.equals("") ? "" : "|") + next))
             .subscribe(System.out::println);
   sleep(5000);
```

```
300|600|900
1200|1500|1800
2100|2400|2700
3000|3300|3600|3900
4200|4500|4800
```

STWB

- **Throttling**
  - The buffer() and window() operators batch up emissions into collections or observables based on a defined scope, which regularly consolidates rather than omits emissions.
    - The throttle() operator, however, omits emissions when they occur rapidly.
    - This is helpful when rapid emissions are assumed to be redundant or unwanted, such as a user clicking a button repeatedly.

STWB

- **Throttling**
  - There are three Observable.interval() sources, the first emitting every 100 milliseconds, the second every 300 milliseconds, and the third every 2,000 milliseconds.
    - Take ten emissions from the first source, three from the second, and two from the third.
    - Use Observable.concat() on all three sources together in order to create a rapid sequence that changes pace at three different intervals

STWB

- **Throttling**

```
public static void main(String[] args) {
    Observable<String> source1 =
         Observable.interval(100, TimeUnit.MILLISECONDS)
                   .map(i -> (i + 1) * 100) //map to elapsed time
                   .map(i -> "SOURCE 1: " + i)
                   .take(10);

    Observable<String> source2 =
         Observable.interval(300, TimeUnit.MILLISECONDS)
                   .map(i -> (i + 1) * 300) //map to elapsed time
                   .map(i -> "SOURCE 2: " + i)
                   .take(3);

    Observable<String> source3 =
        Observable.interval(2000, TimeUnit.MILLISECONDS)
                  .map(i -> (i + 1) * 2000) //map to elapsed t
                  .map(i -> "SOURCE 3: " + i)
                  .take(2);
    Observable.concat(source1, source2, source3)
              .subscribe(System.out::println);
    sleep(6000);
}
```

```
SOURCE 1: 100
SOURCE 1: 200
SOURCE 1: 300
SOURCE 1: 400
SOURCE 1: 500
SOURCE 1: 600
SOURCE 1: 700
SOURCE 1: 800
SOURCE 1: 900
SOURCE 1: 1000
SOURCE 2: 300
SOURCE 2: 600
SOURCE 2: 900
SOURCE 3: 2000
SOURCE 3: 4000
```

STWB

- **Throttling**
  - The last emission at every 1-second interval was all that got through. This effectively samples emissions by dipping into the stream on a timer and pulling out the latest one.

```
Observable.concat(source1, source2, source3)
          .throttleLast(1, TimeUnit.SECONDS)
          .subscribe(System.out::println);
```

```
SOURCE 1: 900
SOURCE 2: 900
SOURCE 3: 2000
```

STWB

- **Throttling**
  - Throttle more liberally at a larger time interval, will get fewer emissions as this effectively reduces the sampling frequency. For example, throttleLast() every 2 seconds

```
Observable.concat(source1, source2, source3)
          .throttleLast(2, TimeUnit.SECONDS)
          .subscribe(System.out::println);
```

```
SOURCE 2: 900
SOURCE 3: 2000
```

STWB

- **Throttling**
  - throttleLast() every 500 milliseconds

```
Observable.concat(source1, source2, source3)
          .throttleLast(500, TimeUnit.MILLISECONDS)
          .subscribe(System.out::println);
```

```
SOURCE 1: 400
SOURCE 1: 900
SOURCE 2: 300
SOURCE 2: 900
SOURCE 3: 2000
```

STWB

- **Throttling**
  - The throttleFirst() operates almost identically to throttleLast(), but it emits the first item that occurs at every fixed time interval

```
Observable.concat(source1, source2, source3)
          .throttleFirst(1, TimeUnit.SECONDS)
          .subscribe(System.out::println);
```

```
SOURCE 1: 100
SOURCE 2: 300
SOURCE 3: 2000
SOURCE 3: 4000
```

STWB

- **throttleWithTimeout**
  - throttleFirst() and throttleLast(), are dissatisfying in one aspect of their behavior.
  - They are agnostic to the variability of emission frequency, and they simply dip in at fixed intervals and pull the first or last emission they find.
  - There is no notion of waiting for a period of silence, where emissions stop for a moment, and that might be an opportune time to push forward the last emission that occurred

STWB

- **Switching**
  - In RxJava, there is a powerful operator called switchMap(). Its usage is similar to flatMap(), but it has one important behavioral difference:
  - it emits the latest Observable derived from the latest emission and disposes of any previous observables that were processing.
    - Cancel an emitting Observable and switch to a new one, thereby preventing stale or redundant processing.

STWB

- **Switching**

```
public static void main(String[] args) {
   Observable<String> items = Observable.just("Alpha", "Beta",
     "Gamma", "Delta", "Epsilon", "Zeta", "Eta", "Theta", "Iota");
   //delay each String to emulate an intense calculation
   Observable<String> processStrings =
        items.concatMap(s -> Observable.just(s)
                .delay(randomSleepTime(), TimeUnit.MILLISECONDS));
    processStrings.subscribe(System.out::println);
    //keep application alive for 20 seconds
    sleep(20000);
}
public static int randomSleepTime() {
   //returns random sleep time between 0 to 2000 milliseconds
    return ThreadLocalRandom.current().nextInt(2000);
}
```

```
Alpha
Beta
Gamma
Delta
Epsilon
Zeta
Eta
Theta
Iota
```

STWB

- **Switching**
  - Create another Observable.interval(), emitting every 5 seconds, and then use switchMap() on it to the Observable to process (which, in this case, is processStrings).
  - Every 5 seconds, the emission going into switchMap() will promptly dispose of the currently processing Observable (if there are any) and then emit from the new Observable it maps to.

STWB

- **Switching**

```
public static void main(String[] args) {
   Observable<String> items = Observable.just("Alpha", "Beta",
     "Gamma", "Delta", "Epsilon", "Zeta", "Eta", "Theta", "Iota");

   //delay each String to emulate an intense calculation
   Observable<String> processStrings =
       items.concatMap(s -> Observable.just(s)
               .delay(randomSleepTime(), TimeUnit.MILLISECONDS));

   Observable.interval(5, TimeUnit.SECONDS)
           .switchMap(i -> processStrings.doOnDispose(() ->
                System.out.println("Disposing! Starting next
           .subscribe(System.out::println);

    //keep application alive for 20 seconds
    sleep(20000);
}
```

```
Alpha
Beta
Gamma
Delta
Epsilon
Zeta
Eta
Disposing! Starting next
Alpha
Beta
Gamma
Delta
Disposing! Starting next
Alpha
Beta
Gamma
Delta
Disposing! Starting next
```

Agenda

- Introduction
- Observables and Observer
- Operators
- Combining Observables
- Multicasting, Replaying and Caching
- Concurrency
- Switching, Throttling, Windowing and buffering
- **Flowables and Backpressure**

Flowables and Backpressure

- **Understanding back pressure**
  - Pushing items synchronously and one at a time from the source all the way to the Observer is indeed how an Observable chain of operators works by default without any concurrency
  - The outputted alternation between Constructing MyItem and Received MyItem shows that each emission is processed one at a time from the source all the way to the terminal Observer.

```
Constructing MyItem 1
Received MyItem 1
Constructing MyItem 2
Received MyItem 2
Constructing MyItem 3
Received MyItem 3
Constructing MyItem 4
Received MyItem 4
Constructing MyItem 5
Received MyItem 5
Constructing MyItem 6
Received MyItem 6
Constructing MyItem 7
Received MyItem 7
...
```

```
public static void main(String[] args) {
    Observable.range(1, 999_999_999)
              .map(MyItem::new)
              .subscribe(myItem -> {
                 sleep(50);
                 System.out.println("Received MyItem " + myItem.id);
              });
 }
```

Flowables and Backpressure

- **Understanding back pressure**
  - Modify previous example and add observeOn(Shedulers.io()) right before subscribe()
  - Note that when MyItem 1001902 is created, the Observer is still only processing MyItem 38. The emissions are being pushed much faster than the Observer can process them

```
public static void main(String[] args) {
    Observable.range( start: 1,  count: 999_999_999) Observa
            .map(MyItem::new) Observable<MyItem>
            .observeOn(Schedulers.io())
            .subscribe(myItem -> {
                sleep( milliseconds: 50);
                System.out.println("Received MyItem " + myItem.id);
            });
}
```

```
...
Constructing MyItem 1001899
Constructing MyItem 1001900
Constructing MyItem 1001901
Constructing MyItem 1001902
Received MyItem 38
Constructing MyItem 1001903
Constructing MyItem 1001904
Constructing MyItem 1001905
Constructing MyItem 1001906
Constructing MyItem 1001907
..
```

Flowables and Backpressure

- **Understanding back pressure**
  - Having two processing threads helps, but, because backlogged emissions get queued by observeOn() in an unbounded manner, this could lead to many other problems, including **OutOfMemoryError** exceptions.

Flowables and Backpressure

- **Flowable**
  - The Flowable class is a variant of the Observable that can tell the source to emit at a pace specified by the downstream operations.
    - In other words, it can exert **backpressure** on the source.

```
public static void main(String[] args) {
    Flowable.range(1, 999_999_999)
            .map(MyItem::new)
            .observeOn(Schedulers.io())
            .subscribe(myItem -> {
                sleep(50);
                System.out.println("Received MyItem " + myItem
            });
    sleep(Long.MAX_VALUE);
}
```

```
Constructing MyItem 1
Constructing MyItem 2
Constructing MyItem 3
...
Constructing MyItem 127
Constructing MyItem 128
Received MyItem 1
Received MyItem 2
Received MyItem 3
...
Received MyItem 95
Received MyItem 96
Constructing MyItem 129
Constructing MyItem 130
Constructing MyItem 131
...
Constructing MyItem 223
Constructing MyItem 224
Received MyItem 97
Received MyItem 98
Received MyItem 99
```

Flowables and Backpressure

- **Flowable**
  - 128 emissions were pushed from Flowable.range() first and 128 MyItem instances were constructed. After that, observeOn() pushed 96 of the constructed items downstream to Subscriber.
  - After these 96 emissions were processed by Subscriber, another 96 were pushed from the source. Then, another 96 were passed to Subscriber

```
Constructing MyItem 1
Constructing MyItem 2
Constructing MyItem 3
...
Constructing MyItem 127
Constructing MyItem 128
Received MyItem 1
Received MyItem 2
Received MyItem 3
...
Received MyItem 95
Received MyItem 96
Constructing MyItem 129
Constructing MyItem 130
Constructing MyItem 131
...
Constructing MyItem 223
Constructing MyItem 224
Received MyItem 97
Received MyItem 98
Received MyItem 99
```

Flowables and Backpressure

- **Flowable**
  - It is almost like the entire Flowable chain strives to have no more than 96 emissions in its pipeline at any given time.
  - Effectively, that is exactly what is happening!
    - This is what we call backpressure, and it effectively introduces a pull dynamic to the push-based operation to limit how frequently the source emits.
  - But why did Flowable.range() start with 128 emissions, and why did observeOn() only send 96 downstream before requesting another 96, leaving 32 unprocessed emissions?
    - The initial batch of emissions is a bit larger, so some extra work is queued if there is any idle time.

Flowables and Backpressure

- **Flowable**
  - Flowable operation started by requesting 96 emissions and continued to emit steadily at 96 emissions at a time, there would be moments where operations might wait idly for the next 96.
  - Therefore, an extra rolling cache of 32 emissions is maintained to provide work during these idle moments, which can yield greater throughput.
  - What is great about Flowable and its operators is that they usually do all the work.
  - No need to specify any backpressure policies or parameters unless custom Flowable needs to be created from scratch.

Flowables and Backpressure

- **Flowable:When to use?**
  - Use Observable if?
    - Expect few emissions over the life of the Observable subscription (fewer than 1,000) or the emissions are intermittent and far apart.
    - Data processing is strictly synchronous and has limited usage of concurrency
    - You want to emit user interface events such as button clicks, ListView selections, or other user inputs on Android, JavaFX, or Swing.
  - Use Flowable if?
    - Dealing with over 10,000 elements
    - Emit from IO operations that support blocking while returning results, which is how many IO sources work.

Flowables and Backpressure

- **Flowable:Flowable and subscriber**
  - On the factory side, there are Flowable.range(), Flowable.just(), Flowable.fromIterable(), and Flowable.interval().
    - Most of these sources support **backpressure**.
  - However, consider **Flowable.interval()**, which pushes time-based emissions at fixed time intervals.
    - If slowed down Flowable.interval(), emissions would no longer reflect the specified time interval and become misleading.
    - Therefore, Flowable.interval() is one of those few cases in the standard API that can throw **MissingBackpressureException**

Flowables and Backpressure

- **Flowable:Flowable and subscriber**

```
public static void main(String[] args) {
    Flowable.interval(1, TimeUnit.MILLISECONDS)
            .observeOn(Schedulers.io())
            .map(i -> intenseCalculation(i))
            .subscribe(System.out::println,
                       Throwable::printStackTrace);
    sleep(Long.MAX_VALUE);
}
```

```
0
io.reactivex.exceptions.MissingBackpressureException:
Cant deliver value 128 due to lack of requests
    at io.reactivex.internal.operators.flowable.FlowableInterval
  ...
```

Flowables and Backpressure

- **Flowable:Flowable and subscriber**
  - Instead of an Observer, **Flowable uses a Subscriber** to consume emissions and events at the end of a Flowable chain.
  - If only lambda event is passed as arguments (and not an entire Subscriber object), subscribe() does not return a Disposable, but rather an org.reactivestreams.Subscription, which can be disposed of by calling **cancel() instead of dispose()**.
  - **Subscription** can also serve another purpose; it communicates upstream how many items are wanted using its **request() method**.
  - Subscription can also be leveraged in the onSubscribe() method of Subscriber, which can call the request() method (and pass in the number of the requested elements) the moment it is ready to receive emissions.

Flowables and Backpressure

- **Flowable:Flowable and subscriber**
  - Just like an Observer, the quickest way to create a Subscriber is to pass lambda arguments to subscribe(). This default implementation of Subscriber requests an unbounded number of emissions, but any operators preceding it still automatically handle the backpressure:

```
public static void main(String[] args) {
    Flowable.range(1, 1000)
            .doOnNext(s -> System.out.println("Source pushed " + s))
            .observeOn(Schedulers.io())
            .map(i -> intenseCalculation(i))
            .subscribe(s ->
                    System.out.println("Subscriber received "
                    Throwable::printStackTrace,
                        () -> System.out.println("Done!")
            );
    sleep(20000);
}
```

```
Source pushed 1
Source pushed 2
...
Source pushed 128
Subscriber received 1
...
Subscriber received 96
Source pushed 129
...
Source pushed 224
Subscriber received 97
```

Flowables and Backpressure

- **Flowable:Flowable and subscriber**
  - Reimplement previous example, but implement our own Subscriber

```
public static void main(String[] args) {
    Flowable.range( start: 1,  count: 1000)
            .doOnNext(s -> System.out.println("Source pushed " + s))
            .observeOn(Schedulers.io())
            .map(i -> intenseCalculation(i))
            .subscribe(new Subscriber<Integer>() {
                @Override
                public void onSubscribe(Subscription subscription) { subscription.request(Long.MAX_VALUE);

                @Override
                public void onNext(Integer s) {
                    sleep( millis: 50);
                    System.out.println("Subscriber received " + s);
                }

                @Override
                public void onError(Throwable e) { e.printStackTrace(); }

                @Override
                public void onComplete() { System.out.println("Done!"); }
            });
    sleep( millis: 20000);
```

Flowables and Backpressure

- **Flowable:Flowable and subscriber**
  - Subscriber to request 40 emissions initially and then 20 emissions at a time after that

```
Flowable.range( start: 1,  count: 1000)
        .doOnNext(s -> System.out.println("Source pushed " + s))
        .observeOn(Schedulers.io())
        .map(i -> intenseCalculation(i))
        .subscribe(new Subscriber<Integer>() {
            2 usages
            Subscription subscription;
            2 usages
            AtomicInteger count = new AtomicInteger( i: 0);

            @Override
            public void onSubscribe(Subscription subscription) {
                this.subscription = subscription;
                System.out.println("Requesting 40 items!");
                subscription.request( l: 40);
            }

            @Override
            public void onNext(Integer s) {
                sleep( millis: 50);
                System.out.println("Subscriber received " + s);
                if (count.incrementAndGet() % 20 == 0 && count.get() >= 40) {
                    System.out.println("Requesting 20 more !");
                }
                subscription.request( l: 20);
            }
```

```
Requesting 40 items!
Source pushed 1
Source pushed 2
...
Source pushed 127
Source pushed 128
Subscriber received 1
Subscriber received 2
...
Subscriber received 39
Subscriber received 40
Requesting 20 more!
Subscriber received 41
Subscriber received 42
...
Subscriber received 59
Subscriber received 60
Requesting 20 more!
Subscriber received 61
Subscriber received 62
...
Subscriber received 79
Subscriber received 80
Requesting 20 more!
Subscriber received 81
Subscriber received 82
```

Flowables and Backpressure

- **Flowable:Using Flowable.create() and BackpressureStrategy**
  - Leveraging Flowable.create() to create a Flowable feels much like Observable.create(), but there is one critical difference: you must specify a **BackpressureStrategy** as a second argument.

```
public static void main(String[] args) {
    Flowable<Integer> source = Flowable.create(emitter -> {
        for (int i = 0; i <= 1000; i++) {
            if (emitter.isCancelled()) {
                return;
            }
            emitter.onNext(i);
        }
        emitter.onComplete();
    }, BackpressureStrategy.BUFFER);
    source.observeOn(Schedulers.io())
          .subscribe(System.out::println);
    sleep(1000);
}
```

```
0
1
2
3
4
...
```

Flowables and Backpressure

- **Flowable:Using Flowable.create() and BackpressureStrategy**
  - This is not optimal because the emissions will be held in an unbounde queue, and it is possible that when Flowable.create() pushes too man emissions, an OutOfMemoryError may result.
  - But at least it prevents MissingBackpressureException and can make your custom Flowable workable to a certain degree

Flowables and Backpressure

- **Flowable:Using Flowable.create() and BackpressureStrategy**
  - There are currently five BackpressureStrategy options you can choose

BackpressureStrategy	Description
MISSING	Essentially results in no backpressure implementation at all. The downstream must deal with backpressure overflow, which can be helpful when used with `onBackpressureXXX()` operators, which we will cover later in this chapter in the *Using onBackpressureXXX() operators* section.
ERROR	Generates `MissingBackpressureException` the moment the downstream cannot keep up with the source.
BUFFER	Queues up emissions in an unbounded queue until the downstream is able to consume them, but can cause an `OutOfMemoryError` if the queue gets too large.
DROP	If the downstream cannot keep up, this ignores upstream emissions and does not queue them while the downstream is busy.
LATEST	This keeps only the latest emission until the downstream is ready to receive it.

Flowables and Backpressure

- **Flowable:Flowable to Observable and back**
  - Turn an Observable into Flowable by calling its toFlowable() operator, which accepts a BackpressureStrategy as an argument. In the following code, we turn Observable.range() into Flowable using BackpressureStrategy.BUFFER.

```
public static void main(String[] args) {
    Observable<Integer> source = Observable.range(1, 1000);
    source.toFlowable(BackpressureStrategy.BUFFER)
          .observeOn(Schedulers.io())
          .subscribe(System.out::println);
    sleep(10000);
}
```

Flowables and Backpressure

- **Flowable:Flowable to Observable and back**
  - Flowable also has a toObservable() operator, which will turn a Flowable<T> into an Observable<T>.
  - This can be helpful in making a Flowable usable in an Observable chain, especially with operators such as flatMap()

```
public static void main(String[] args) {
    Flowable<Integer> integers =
            Flowable.range(1, 1000)
                    .subscribeOn(Schedulers.computation());
    Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
              .flatMap(s -> integers.map(i -> i + "-" + s)
                                    .toObservable())
              .subscribe(System.out::println);
    sleep(5000);
}
```

Flowables and Backpressure

- **Flowable:onBackpressureXXX**
  - The onBackPressureBuffer() takes an existing Flowable that is assumed to not have backpressure implemented and applies BackpressureStrategy.BUFFER at that point to the downstream.

```
public static void main(String[] args) {
    Flowable.interval(1, TimeUnit.MILLISECONDS)
            .onBackpressureBuffer()
            .observeOn(Schedulers.io())
            .subscribe(i -> {
                sleep(5);
                System.out.println(i);
            });
    sleep(5000);
}
```

Flowables and Backpressure

- **Flowable:onBackpressureXXX**
  - Hold a maximum capacity of 10 and specify the BackpressureOverflowStrategy.DROP_LATEST strategy for an overflow

```
public static void main(String[] args) {
    Flowable.interval(1, TimeUnit.MILLISECONDS)
            .onBackpressureBuffer(10,
                    () -> System.out.println("overflow!"),
                    BackpressureOverflowStrategy.DROP_LATEST)
            .observeOn(Schedulers.io())
            .subscribe(i -> {
                sleep(5);
                System.out.println(i);
            });
    sleep(5000);
}
```

```
...
overflow!
overflow!
135
overflow!
overflow!
overflow!
overflow!
overflow!
136
overflow!
overflow!
overflow!
overflow!
overflow!
492
overflow!
overflow!
overflow!
...
```

Flowables and Backpressure

- **Flowable:onBackpressureXXX**
  - Notice that there is a jump between 127 and 494. This is because all numbers in between were ultimately beaten by 494 being the latest value and, at that time, the downstream was ready to process more emissions.

```
public static void main(String[] args) {
    Flowable.interval( period: 1, TimeUnit.MILLISECONDS)
            .onBackpressureLatest()
            .observeOn(Schedulers.io())
            .subscribe(i -> {
                sleep( millis: 5);
                System.out.println(i);
            });
    sleep( millis: 5000);
}
```

```
...
122
123
124
125
126
127
494
495
496
497
...
```

Flowables and Backpressure

- **Flowable:onBackpressureXXX**
  - You can optionally provide an onDrop lambda argument, specifying what to do with each dropped item, as shown in the following code (the dropped items are simply printed

```
public static void main(String[] args) {
    Flowable.interval( period: 1, TimeUnit.MILLISECONDS)
            .onBackpressureDrop(i ->
                    System.out.println("Dropping " + i))
            .observeOn(Schedulers.io())
            .subscribe(i -> {
                sleep( millis: 5);
                System.out.println(i);
            });
    sleep( millis: 5000);
}
```

```
...
Dropping 653
Dropping 654
Dropping 655
Dropping 656
127
Dropping 657
Dropping 658
Dropping 659
Dropping 660
Dropping 661
493
Dropping 662
Dropping 663
Dropping 664
...
```

Flowables and Backpressure

- **Flowable:generate**
  - Although the standard Flowable factories and operators automatically handle the backpressure, the onBackPressureXXX() operators, while quick and effective for some cases, just cache or drop emissions, which is not always desirable.
    - It would be better to force the source to slow down as needed in the first place.
  - Thankfully, Flowable.generate() exists to help create backpressure, respecting sources at a nicely abstracted level.
    - It accepts a Consumer<Emitter<T>>, much like Flowable.create(), but uses a lambda to specify which onNext(), onComplete(), and onError() events to pass each time an item is requested from the upstream.

Flowables and Backpressure

- **Flowable:generate**
  - The simplest overload for Flowable.generate() accepts just Consumer<Emitter<T>> and assumes that there is no state maintained between emissions.
    - This can be helpful in creating a **backpressure-aware** random integer generator

```
public static void main(String[] args) {
    randomGenerator(1, 10000)
            .subscribeOn(Schedulers.computation())
            .doOnNext(i -> System.out.println("Emitting " + i))
            .observeOn(Schedulers.io())
            .subscribe(i -> {
                sleep( millis: 50);
                System.out.println("Received " + i);
            });
    sleep( millis: 10000);
}

1 usage
static Flowable<Integer> randomGenerator(int min, int max) {
    return Flowable.generate(emitter ->
            emitter.onNext(ThreadLocalRandom.current().nextInt(min, max))
    );
}
```

```
...
Emitting 8014
Emitting 3112
Emitting 5958
Emitting 4834   // 128th emission
Received 9563
Received 4359
Received 9362
...
Received 4880
Received 3192
Received 979    // 96th emission
Emitting 8268
Emitting 3889
Emitting 2595
...
```